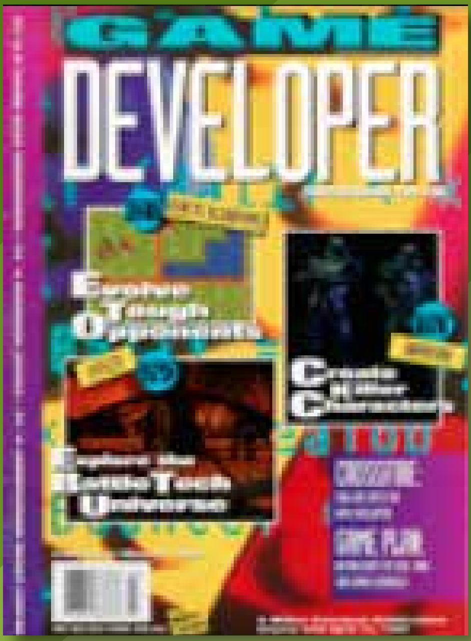
gd

GAME DEVELOPER MAGAZINE

FEBRUARY/MARCH 1996



UBM Tech

Back

Legitimate Concerns

I recently spent some time in southern Arizona, an area with dark skies, low humidity, and still air. Throw in some high desert and mountains to give a few thousand feet more elevation and it's understandable that the area is a hotbed for astronomy, both professional and amateur.

As with many of you, science fiction was the staple of my literary diet from third grade on, until it seemed that I'd read everything from Asimov to Zelazny (Poul Anderson may come before Asimov in the alphabet, but not in my heart). Those stories of starship troopers, Bene Gesserit, lensmen, K'Zinti, and their like, became part of what I saw when I looked at the night sky.

Since I grew up in pretty much urban areas, I was mostly confined to stars of the first few magnitudes, the moon, and the more obvious planets. Nevertheless, it was always my conviction that if I could just see well enough, I would see a universe filled with ships flying on solar sails, generation ships carved from asteroids, fleets of interstellar transports picking their way through foggy nebulas, and stars winking out as the last section of their Dyson spheres were nailed into place.

Recent photos from the Hubble Space Telescope reminded me of those dreams. Recent images from the Great Nebula of Orion show “cosmic eggs,” their three-dimensional structure perfectly lit by conveniently placed protostars. The most spectacular space effects from Pixar and Industrial Light and Magic have nothing on these photos for sheer beauty. So when I was in Arizona, I made sure that I was introduced to someone with an “amateur” telescope, an 11-inch Schmidt-Cassegrain with worm-gear corrected equatorial mount and a computerized tracker containing virtually the entire Messier catalog of deep-space objects. Of course, I knew I was not going to see anything similar to the HST photos, but I did hope that on a clear night with no moon, I would see my first nebula.

The photos in *Sky & Telescope* magazine don't do modern amateur telescopes justice. Ed, the scope's owner, has a special indoor/outdoor room to house his equipment. He moves the 11-inch scope with a hand-truck out onto his patio. Tucson is a town remarkably conscious of light pollution due to its proximity to the Kitt's Peak observatory, and excellent viewing can be had in the suburbs. After setting up the telescope in rough approximation with Polaris, Ed centered Deneb in the tracking scope and punched a button or two on the mount's control panel.

A few minutes later, the computer had seen enough to do three-dimensional transformations worthy of Chris Hecker to calculate its exact orientation relative to the Earth's axis. At that point, the computer and precision motors kept objects only a few arcseconds wide locked in the center of the view, despite the Earth's rotation. A charge-coupled device (CCD) can give the even more precise control necessary for long-exposure photography.

The computer handles one more task. Ed types in a Messier number and rotates the tube until the LED display indicates precise alignment. Without looking through the spotting scope, he invites me to take a look. “That's the Crab Nebula,” he says. Chinese astronomers have precise records of the appearance of the nova that created the nebula, an explosion so epochal that it could be seen in daylight. I look out and back into time. The nebula appears as a faint puff of smoke, gray against black. Not even a thousand years after the explosion, the remains of the star are spread over light years. Roughly 6 trillion miles in a light year, cubed is 256 times 10 to the 12th to the 3rd, so the volume of a single light year is roughly, uh, really, really big. But

GAME DEVELOPER

Editor Larry O'Brien
gdmag@mfi.com

Senior Editor Nicole Freeman
76702.706@compuserve.com

Managing Editor Nicole Claro
nclaro@mfi.com

Production Editor Diane Anderson
dianderson@mfi.com

Editorial Assistant Jana Outlaw
joutlaw@mfi.com

Contributing Editors Alex Dunne
76702.1142@compuserve.com
Barbara Hanscome
bhanscome@mfi.com
Chris Hecker
checker@bix.com
Mike Michaels
mike@irvine.com
David Sieks
dsieks@arnarb.harvard.edu

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

Cover Photography Charles Ingram Photography

Publisher Veronica Costanza
Group Director Regina Starr Ridley

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

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

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

there it is, rock steady in the eyepiece, a structure so vast I can't even tame it with exponents.

Ed has a flare for the dramatic and turns to the other extreme, the Great Nebula of Orion, and shows me another faint puff of smoke. He makes sure I note some of the stars embedded in the nebula and tells me I'm seeing the birth flares of stars newborn just a million years ago. The HST photos were of this area and a thousand times more detailed and colorful, but there is something far more powerful about the ancient light falling directly on my retina.

Finally, Ed shows me the Andromeda Galaxy. It's 250,000,000 light years away, but it's so large that it doesn't even fit in the field of view. The extended shape of the galaxy is apparent, but no structure—like the nebulas, it appears as a blob of grey that otherwise might be taken for a lens smudge. But to me, it instantly brings back those childhood dreams of worlds like grains of sand. An entire galaxy glowing as a homogenous whole from its stars.

The point I'm trying to get at is that many of the most powerful experiences available to us are not inherently grandiose. If there are two experiences and one is somehow felt as "more legitimate," it won't matter if the other is more detailed, longer lasting, or more comfortable. Human eyes will never see the colors and details of an astronomical photograph. But looking through a telescope is better. Any night on The Discovery Channel, I can see better images of sharks and whales than I've ever seen underwater. But I'd never trade a single dive for all the film in the world.

As designers of digital entertainment, we have to always remind ourselves that we will never create experiences as physically legitimate as the simplest activity. There is not a fighting game that compares to capture the flag, not a sports game that compares to dodge ball, not a strategy game that compares to cutting over on the frontage road to avoid traffic. So we must create our legitimacy from game play. We must never smirk at the universe we create (unless our universe does nothing but smirk), we must never wink at the player and say, "We both know this couldn't happen, but you'll forgive me because we both know this is just a computer game."

Too many games rely on just such tactics, relying on either AI that cheats or stories with very limited branching. Game design, AI, and story-telling will be major themes of *Game Developer* in 1996, and we're excited to kick off our third year of publication with a major article on genetic algorithms, beginning on page 26.

When we are done, Ed trundles his telescope back into its private room. As I walk away, I wonder what chip powered its computer. A Z-80? An 8086? Who says you need 32 bits to have fun? ■

Larry O'Brien
Editor

Go On! (Line, That Is)

Alex Dunne

You need to stay on top of things to make it as a game developer. Here, Alex Dunne gives you some research assistance—he's hunted down some URLs you simply cannot afford to miss.

If you've been scanning the bookshelves and magazine racks, you've noticed that the number of words devoted to game programming isn't quite on par with the demand generated by developers. Information on developing games is sparse. There is more information available to developers of web sites, general business applications, and the like. If you're like many others in this industry, you've either begun to scan web sites and ftp archives and online services such as CompuServe, America Online, and Microsoft Network for programming information or you're about to.

Unfortunately, the wired world moves in mysterious ways, and trying to locate current, pertinent information on the World Wide Web and online services can be a time-consuming experience. It's nice to have directions before setting out on this road. I've pieced together my favorite sites from long hours of browsing, which will hopefully save your eyes and your telephone bill from ruin. We'll be transferring this information to the *Game Developer* web site shortly (probably by the time you read this), so don't worry about copying down the long URLs.

If you come across some new web or ftp sites that you'd like to let other developers in on, let us know. Our goal is to turn our own web site into a launching point for other information on the Internet, and we'll need your help to keep our links hot and fresh. Send your recommendations to gdmag@mfi.com with the subject line "Hot URL."

The World Wide Web

As evidenced by Netscape's ridiculous stock price, the web is growing so fast we can't keep up with it. For this reason, the information you find on the net is both fresh as this morning's Starbucks and as old as that tin of Folger's Crystals in the back of your pantry. As many web pages went up yesterday as the number of pages that haven't been updated in the past year. So the web can be an extremely efficient means of getting the information you want, or a time-consuming chase across the world that terminates at an antiquated site. Here are some good sites to visit for game development topics.

- *3D Graphics Engines page (http://www.cs.tu-berlin.de/~ki/engines.html)* A fairly fleshed-out list of texture mapping engines, Gouraud shading engines, landscape rendering engines, flat-shading engines, and more. A bit like a buyers guide, with links to vendor home pages.
- *Algorithm's Graphics Hotlist (http://www.algorithm.com/graphics/graphhot.html)* This contains links to all sorts of graphics-related pages and, like the Virtual Library, is a good launching point.
- *Computer Game Developers Conference (CGDC) (http://www.mfi.com/sdconfs/cgdc)* Information about the West Coast version of the game developer's conference. Conference schedule, location, and more.
- *Computer Graphics Miscellaneous FAQ (http://www.algorithm.com/graphics/graphhot.html)* A list of frequently asked questions ("How do

I..”; “Where can I find…”) about graphics as well as pointers to other sites. Updated fairly regularly.

- *Denthor of Asphyxia and the Asphyxia Trainers (http://goth.vironix.co.za/~denthor/)* Don’t ask me about the name, but this site offers about 20 downloadable tutorials, such as “widgets and gismos,” “crossfading,” “fullscreen scrolling,” “starfields,” “pixel morphs,” “scaling bitmaps,” and more.
- *Egerter Software (http://peace.wit.com/~kosmic/gooroo/software.html)* A small Canadian software company posting three graphics programming tutorials and two rendering engines.
- *Ethan Brodsky’s page. (http://www.xraylith.wisc.edu/~ebrodsky/)* A source of information for programming Sound Blaster audio cards.
- *Game Resource Page (http://www.cs.umu.se/~christer/GR/)* A directory of game programming sites divided into algorithms, graphics, FAQs, book publishers, and more.
- *Game Developers Contact List (http://www.gamesdomain.co.uk/gamedev/reslist.html)* Conceptually a great idea, but it could be updated more frequently (it’s two months old as I write this). A clearing house of people interested in various areas of game development (writers, programmers, musicians) who want to get matched up with others in the field. A bit like the personals section of your local newspaper.
- *Knut’s Homepage (http://www.oslonett.no/home/oruud/homepage.htm)* This page contains information about game programming and graphics. It also contains links to shareware and freeware graphics and sound libraries available on the Internet.
- Game Developer *magazine (http://www.mfi.com/gdmag)* I’ll be immodest and recommend that you check out our site. Up to this point, we’ve been tentative about making a big splash on the web because our resources are fairly stretched just producing a print version of the magazine. But we’re revamping our web site and will be relaunching it with a new interface and new content.
- *Game Development Tools Listing (http://www.cs.umu.se/~christer/GR/game_company_page.html)* This page lists company web sites broken down by product category, such as “Entertainment Producers,” “Hardware Manufacturers,” and “Multimedia Software Manufacturers.” Listed alphabetically. Very handy.
- *Nexus Game Programming Links Page (http://www.gamesdomain.co.uk/gamedev/gprog.html)* Based out of the U.K., a well-maintained site for “all things game-related on the Internet.”
- *Searching: DejaNews (http://www.dejanews.com)* This is an incredible site that searches for Usenet newsgroups based on keywords you enter. Looking for every mention of the term “texture mapping” in rec.games.programmer? Alternatively, check to see if anyone’s been talking about you online.
- *Searching: Savvy Search (http://www.cs.colostate.edu/~dreiling/smartform.html)* A search engine that sends your query in parallel to other search engines including Yahoo, Lycos, Web Crawler, and NIKOS (about 15 in all) and displays the results in a single list.
- *Searching: Yahoo’s Search Engine Master List (http://www.yahoo. com/Computers_and_Internet/Internet/World_Wide_Web/Searching_the_Web/)* This lists most of the search engines available on the web. Take your pick.
- *Watcom Corporation (http://www.watcom.on.ca/c/c.html)* Information about C and C++. Watcom has downloadable tutorials for those learning the language, links to other C/C++ resources, and a library of source code.
- *WWW Virtual Library of Computer Graphics (http://www.dataspace.com/WWW/vlib/comp-graphics.html)* If you want a comprehensive list of papers, organizations, and commercial sites dedicated to graphics, this is a good place to start.

File Transfer Protocol (FTP) Sites

These sites allow you to download files (usually zipped—make sure you’ve got PKunzip handy) for offline viewing, inspection, and cut and paste (careful with the copyrights though).

- *Borland’s FTP Site (ftp://ftp.borland.com/pub/techinfo/index.html)* Whether you’re looking for C, C++, or Pascal source code or support, Borland’s put together a comprehensive site that’s logically organized.
- *Creative Labs’s FTP site (ftp://ftp.creaf.com)* Sound files, utilities, and the like.
- *DEC’s FTP Site (http://ftp.digital.com/cgi-bin/grep-index)* Digital’s has a large library of source code for a number of games. Most of the code isn’t that fresh, but nevertheless it’s interesting to browse.
- *Fastgraph FTP Site (ftp://ftp.accessnv.com/fg/)* This is where you can find the shareware version of the Fastgraph graphics library (for PC). There are also articles and chapters from *Action Arcade Adventure Set* (Coriolis Group, 1994).
- Game Developer *magazine (ftp://ftp.mfi.com/pub/gamedev/src/)* You can find the source code from our back issues at this location.
- *Game Programming Archives at Oulu University, Finland. (ftp://x2ftp.oulu.fi/pub/msdos/programming/)* This is a huge site with many different files for downloading, ranging in topic from graphics to AI, game programming theory, sound, and much more. Many web directories have links to this source, as it’s very comprehensive. Word has it that it’s not being updated anymore, however.
- *Microsoft FTP site (ftp://ftp.microsoft.com/dirmap.htm)* This page helps you navigate through the myriad files found at Microsoft.
- *Mode X Introduction (ftp:// x2ftp.oulu.fi/pub/msdos/programming/docs/xintro18.zip)* Self-explanatory, eh?

Online Service: CompuServe!

I admit I’m heavily biased toward CompuServe. I’ve tried America Online and the Microsoft Network, and found both of them extremely lacking in technical information. CompuServe, on the other hand, has a fairly rich offering of technical forums, hosted by magazines and software and hardware vendors. What makes these forums so valuable, however, isn't these corporate sponsors, but the CompuServe members who hang out in these areas. Most have technical backgrounds, contribute frequently to forum libraries, participate in discussions, and answer your questions. These forums are moderated either by employees of the sponsor or other citizens, such as yourself, which helps filter out spams, flames, and other noise. If you only belong to one online service, this is the one. Here are some forums to visit:

- *Autodesk Multimedia Forum (GO ADESK)* has project/mesh files for 3D Studio and support for both 3D Studio and Animator Pro. If you're using either of these Autodesk tools, this is your spot.
- *CASE Forum (GO CASEFO)* is a forum where issues such as software engineering, software quality, and development teamwork are discussed.
- *Dr. Dobb's Journal Forum (GO DDJ-FORUM)* is sponsored by the magazine of the same name. It contains libraries on C/C++, artificial intelligence, and of course code from the magazine itself.
- *Game Developer's Forum (GO GAMEDEV)* contains good discussions and searchable library sections on Windows programming, DOS programming, online and modem games, design theory, hardware issues, and more. Game kits, sample code, and FAQ sheets galore. Advice on tools and techniques.
- *Game Players' Forum (GO GAMERS)* is a forum for game players. It contains comments about games, cheat files and codes, and so on. A good barometer of what players like and hate.
- *Microsoft Multimedia (GO WINMM)* is the official Microsoft forum for multimedia and game development with Windows.
- *Software Development Forum (GO SDFORUM)* contains source code from *Game Developer* magazine and sections devoted to C++, Visual Basic, object-oriented programming, and more.
- *Visual Basic Forums (GO VBPJ, GO MSBASIC)* are two forums useful for VB programmers.

CompuServe has a rich offering of technical forums, hosted by magazines and hardware and software vendors.

Frequently Asked Questions (FAQs)

Do you want to learn about a particular topic, such as 3D programming, but you don't know the right questions to ask? Try a FAQ sheet. Many FAQ sheets describe Usenet newsgroups for which they were created, but they're more than just newsgroup descriptions. Questions that keep getting asked in these newsgroups get added to the newsgroup FAQs, and if you are new to a newsgroup, you are often instructed to read the group's FAQ before you post any questions. That saves bandwidth and prevents people from having to answer a question that might have been answered ten— or 100—times already.

- *3D programming information (ftp://x2ftp.oulu.fi/pub/msdos/programming/faq/3d-prog.18)*
- *BSP tree FAQ (http://www.qualia.com/bspfaq/)*
- *Getting Started in Game Development FAQ (ftp://ftp.accessnv.com/fg/misc/gamefaq.txt)*
- *comp.graphics FAQ (ftp://x2ftp.oulu.fi/pub/msdos/programming/faq/graphics.faq)*
- *comp.graphics.algorithms FAQ (http://www.cis.ohio-state.edu/hypertext/faq/usenet/graphics/algorithmsfaq/faq.html)*
- *comp.graphics.animation FAQ (ftp://x2ftp.oulu.fi/pub/msdos/programming/faq/animatio.12)*
- *comp.lang.c FAQ (ftp://rtfm. mit.edu/pub/usenet/news.answers/C-faq/faq)*
- *rec.games.programmer FAQ (http://www.ee.ucl.ac.uk/~phart/game/FAQ/rgp_FAQ.html)*
- *rec.games.design FAQ (ftp://x2ftp.oulu.fi/pub/msdos/programming/faq/design.201)*
- *Mode X FAQ (ftp://x2ftp.oulu.fi/pub/msdos/programming/faq/modex.faq)*
- *PC soundcards FAQ (http://www.cis.ohio-state.edu/hypertext/faq/usenet/PCsoundcards/soundcard-faq/faq.html)*
- *Tile-based games FAQ (http://www.io.com/~paulhart/FAQ/tilefaq.html)*

Open the Floodgates...

You probably know of a number of sites that I haven't mentioned here. Hey, for that matter, so do I! Unfortunately, space constraints don't allow me to print every worthy site. For that reason, we'll be extending this list and posting it on our own web site. Start sending us your URLs! ■

Alex Dunne is a contributing editor to Game Developer *magazine. Contact him via e-mail at 76702.1142@compuserve.com.*

Back

Shocking the Monkey

Diane Anderson

Find out what the Internet may mean for game programming—it can be "powered by Shockwave" nowadays. And see who's upgrading what, when, and how much it'll cost.

When you visit the *Toy Story* site, you can see the delightful characters created by Pixar. Woody and Buzz Lightyear have a game—simple and silly, but a game nonetheless. It's a game developed using the latest Director technology—a memory game, you know, like the Husker Du game you played as a child in which you turn over a card and try to find a match? Well, Shockwave users get to turn over cards of *Toy Story* characters and search for matches. Okay, so it isn't an intellectually gruelling game of chess. But this new technology should mean some big things for interactive game programming. It is definitely worth the time it takes to examine what Macromedia is doing with Director and Shockwave. Check out Macromedia's web site at http://www.macromedia.com. If you have the right version (which happens to be 2.0 at the exact moment of this writing) of Netscape and are running Windows 95, you won't be bored by a static screen. You can see what's happening at Melrose Place, c/net, DC Comics, Intel, MTV, CNN, and the American Cancer Society. Your monitor will be animated, which may not be new, but coming over the Internet, it is pretty exciting. Check it out and then get started making interactive games for the net yourself.

Get Your Macros Here

The Macromedia folks are up to other things besides Shockwaving their programs. While Afterburner and Shockwave might seem cool enough to the average bear, Macromedia has introduced Extreme 3D for cross-platform modeling, rendering, and animation. It uses Director to provide advanced data filtering, visual feedback, interactive keyframe manipulations, and cut and paste in a single window. It costs $699. (They also agreed to acquire OSC, makers of digital audio production software.)

- For more information contact:
  Macromedia
  600 Townsend St.
  San Francisco, Calif. 94103
  Tel: (415) 252-2000
  Fax: (415) 626-1502

Is 3D Space True?

Caligari Corp. announced its 3D product, TrueSpace2. TrueSpace2 uses Intel's 3DR acceleration to let the user create and render with drag-and-drop capabilities. Users can perfom 3D Boolean operations and model scenes. TrueSpace2 has PostScript capabilities in which the user loads 2D drawings in an .EPS format into 3D spaces. The program for Windows costs $499, but you can order a free CD-ROM trial version if you agree to fork over $14.95 to cover shipping and handling.

- For more information contact:
  Caligari Corp.
  1955 Landings Dr.
  Mountain View, Calif. 94043
  Tel: (415) 390-9600
  Fax: (415) 390-9755

Schmooze news...

Softkey yanked the rug out from under Broderbund's acquisition of The Learning Company with a hostile takeover bid. They went on to complete their edutainment triple play by buying Compton's New Media and Mecc (makers of Oregon Trail). Will these companies experience the fate of Softkey company WordStar—now just another low-cost retail relicensor? According to James Sigismonti, Compton's Senior Producer, Softkey says it wants Compton's to keep up its quality development. But Softkey shareware developers allege Softkey has refused to pay them. Can Softkey, who is closing in on the #2 spot in software sales, change its low-end image?Are developers at the hands of miserly monopolies?

And from the #1 software company, MicroSoft announced they bought Bruce Artwick's company, developer of the highly successful MicroSoft Flight Simulator. MicroSoft cofounder, Paul Allen purchased 5% of Broderbund stock for his own use.

Larry Kasanoff's Threshold Entertainment, creators of the Mortal Kombat movie, secured film rights to both The Seventh Guest and 11th Hour, brainchildren of Trilobyte.

Sources at MicroSoft indicate that Direct Draw in on track with the recent ship on MSDN. Direct Play still has a long way to go before being really useful. Direct Sound is in much better shape. Direct 3D should be in the hands of developers early next year with the current build fast, but not fast enough yet.

Technologist Steve Crane, recently with Knowledge Adventure, has joined Activision. Senior Producer Leonard Mlodinow, of Knowledge Adventure, is now bringing his sense of humor to Disney Interactive. Jeff Dee, formerly Art Director with Simtex, has joined Illusion Machines Incorporated. Josh Davidson, producer for Microsoft, has moved to Dreamworks Interactive in L.A. as part of the Microsoft-Dreamworks joint venture. Jason Rubenstein, formerly of the ImagiNation Network, has joined Dreamworks Interactive's marketing. Geoffrey Selzer has joined Disney Interactive. Executive Producer Lisa Linnenkohl, mostly recently with Star Press Multimedia, has joined Palladium Interactive.

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

Blast it!

Creative Technology Ltd. (Creative Labs) announced a new version of its 3D game board, 3D Blaster for 486 VL-BUS PCs. Now there's Sound Blaster, Phone Blaster, Modem Blaster, *and* 3D Blaster. Creative seems to be working with Microsoft to bring 3D to every Pentium. 3D Blaster will be available this spring and should retail for about $349; it will be bundled with software titles, like Magic Carpet, that were created using Creative Labs's product.

■ For more information contact:
Creative Labs
1901 McCarthy Blvd.
Milpitas, Calif. 95035
Tel: (408) 428-6600
Fax: (408) 428-2394

Use a Tool

The company formerly known as HSC Software is now renamed to be known as MetaTools Inc. MetaTools has announced that Kai Power Tools and KPT Convolver can now plug into Autodesk's Animator Studio. These extensions for Windows let animators blend, texturize, and saturate their animations. These imaging tools let you create gradients, fractals, transparency effects, explosion effects, particle effects, moving bubbles, and spheres. You can adjust hue, sharpness, contrast, relief, and edges of an object or area using the KPT Convolver.

■ For more information contact:
MetaTools Inc.
6303 Carpinteria Ave.
Carpinteria, Calif. 93013
Tel: (805) 566-6200
Fax: (805) 566-6385
Web: http://www.metatools.com

Engine Engine

Number Nine Visual Technology announced its 128-bit graphics accelerator, Imagine 128 Series 2. Imagine 128 Series 2 has a built-in 256-bit video engine and is capable of speedy generation of Gouraud-shaded lines and triangles, Z buffering, spatial blending, and decal-mode texture mapping. It can handle three operand bit `blt`s at two operand speeds and has a hardware DIB engine. Imagine 128 Series 2 costs $399.

■ For more information contact:
Number Nine Visual Technology
18 Hartwell Ave.
Lexington, Mass. 02173
Tel: (617) 674-0009
Fax: (617) 674-2919

UPGRADE YOURS!

- You've seen Electric Image in things like *Congo, The Net*, *Batman Forever*, *Jurassic Park*, and *Star Trek*. You've seen what it can do in many of your favorite games. You've even seen it in action in those Intel commercials (you know, the ones with the flying Pentiums and dolphins?). Well now Electric Image has released Electric Image 2.5.2. PICT and QuickTime files can be used as maps with the newest version. Electric Image lets animators finesse Inverse Kinematics animation and render directly into the Quicktime animation codec format. It costs $990 to upgrade from version 2.0 to 2.5.2 and $495 to upgrade from 2.1. to 2.5.2. If you are buying it for the first time, 2.5.2 costs $7,495.

- Fractal Design announced the newest version of its image editor, Painter 4. Designers can now blend raster and vector imagery and collaborate on artwork over LANs and the Internet. It works with Photoshop, Illustrator, and Freehand. It runs on Windows and Macintosh and costs $549.

Diane Anderson is managing editor of Game Developer. *Contact her at dianderson@mfi.com.*

Back

Let's Get to the (Floating) Point

Chris Hecker

Question: Is floating-point math inherently evil? Answer: Maybe not, if the instruction timings of modern chips are to be believed. Question: But are they to be believed?

When I sat down to write this article, it was supposed to be the last installment in our epic perspective texture mapping series. Part of the way through, I realized I needed to cover what ended up as the actual topic of this article—floating-point optimizations—to complete the optimizations on the perspective texture mapper's inner loop. So we'll learn some generally cool tricks today and apply them to the texture mapper in the next issue. In fact, these techniques are applicable to any high-performance application that mixes floating-point and integer math on modern processors. Of course, that's a long and drawn-out way of saying the techniques are eminently suitable to cool games, whether they texture map or not.

The Real Story

A few years ago, you couldn't have described a game as an "application that mixes floating-point and integer math," because no games used floating-point. In fact, floating-point has been synonymous with slowness since the beginning of the personal computer, when you had to go out to the store, buy a floating-point coprocessor, and stick it into a socket in your motherboard by hand. It's not that game developers didn't want to use real arithmetic, but the original PCs had enough trouble with integer math, let alone dealing with the added complexities of floating-point.

We don't have enough space to cover much of the history behind the transition from integer math, through rational (Bresenham's line-drawing algorithm, for example) and fixed-point arithmetic, and finally to floating-point, but here's the quick summary: For a long time, game developers only used floating-point math in prototype algorithms written in high-level languages. Once prototyped, the programmers usually dropped the code down into fixed-point for speed. These days, as we'll see, floating-point math has caught up with integer and fixed-point in terms of speed and in some ways has even surpassed it.

To see why, let's look at the cycle timings of the most common mathematical operations used by game developers (addition, subtraction, multiplication, and—hopefully relatively rarely—division) for fixed-point, integer, and floating-point. Table 1 shows the cycles times on three generations of Intel processors, the PowerPC 604, and a modern MIPS. We see that the floating-point operations, with the exception of additions and subtractions, are actually faster than their integer counterparts on the modern processors (the 386, a decidedly nonmodern processor without an integrated floating-point unit, lags far behind, and the transitional 486 has mixed results).

Of course, these numbers alone don't tell the whole story. The table doesn't show that, although the cycle times are still slow compared to single-cycle instructions like integer addition, you can usually execute other integer instructions while the longer floating-point operations are running. The amount of cycle overlap varies from processor to processor, but on really long instructions, like floating-point

Table 1. Various Instruction Timings (Parentheses Indicate Single Precision)

	Integer Add/Subtract	Float Add/Subtract	Integer Multiply	Float Multiply	Integer Divide	Float Divide
Intel 386/387	2	23-34	9-38	27-35	43	89
Intel i486	1	10	12-42	11	43	62 (35)
Intel Pentium	1	3	10	3	46	33 (19)
PowerPC 604	1	3	4	3	20	31 (18)
MIPS R4x00	1	4	10	8 (7)	69	36 (23)

division, you can usually overlap all but a few cycles with other integer (and sometimes even floating-point) instructions. In contrast, the longer integer instructions allow no overlap on the current Intel processors and limited overlap on the other processors.

On the other hand, the floating-point operations are not quite as fast as Table 1 implies because you have to load the operands into the floating-point unit to operate on them, and floating-point loads and stores are usually slower than their integer counterparts. Worse yet, the instructions to convert floating-point numbers to integers are even slower still. In fact, the overhead of loading, storing, and converting floating-point numbers is enough to bias the speed advantage towards fixed-point on the 486, even though the floating-point instruction timings for the actual mathematical operations are faster than the corresponding integer operations.

Today, however, the decreased cycle counts combined with the tricks and techniques we'll cover in this article give floating-point math the definite speed advantage for some operations, and the combination of floating-point and fixed-point math is unbeatable.

If It Ain't Float, Don't Fix It

As usual, I'm going to have to assume you know how fixed-point numbers work for this discussion. Mathematically speaking, a fixed-point number is an integer created by multiplying a real number by a constant positive integer scale and removing the remaining fractional part. This scale creates an integer that has a portion of the original real number's fraction encoded in its least significant bits. This is why fixed-point was the real number system of choice for so many years; as long as we're consistent with our scales, we can use fast integer operations and it just works, with a few caveats. It has big problems with range and is a mess to deal with, for the most part. You need to be very careful to avoid overflow and underflow with fixed-point numbers, and those "fast" integer operations aren't as fast as the same floating-point operations anymore.

Floating-point math, on the other hand, is a breeze to work with. The main idea behind floating-point is to trade some bits of precision for a lot of range.

For the moment, let's forget about floating-point numbers and imagine we have really huge binary fixed-point numbers, with lots of bits on the integer and fractional sides of our binary point. Say we have 1,000 bits on each side, so we can represent numbers as large as 2^{1000} and as small as 2^{1000}. This hypothetical fixed-point format has a huge range and a lot of precision, where range is defined as the ratio between the largest and the smallest representable number, and precision is defined as how many significant digits (or bits) the representation has. So, for example, when we're dealing with incredibly huge numbers in our galaxy simulator, we can still keep the celestial distances accurate to within subatomic particle radii.

However, most applications don't need anywhere near that much precision. Many applications need the range to represent huge values like distances between stars or tiny values like the distance between atoms, but they don't often need to represent values from one scale when dealing with values of the other.

Floating-point numbers take advantage of this discrepency between precision and range to store numbers with a very large range (even greater than our hypothetical 2,000-bit fixed-point number, in fact) in very few bits. They do this by storing the real number's exponent separately from its mantissa, just like scientific notation. In scientific notation, a number like 2.345 x 10^{35} has a mantissa of 2.345 and an exponent of 35 (sometimes you'll see the terms significand and characteristic instead of mantissa and exponent, but they're synonymous). This number is only precise to four significant digits, but its exponent is quite big (imagine moving the decimal point 35 places to the right and adding zeros after the mantissa runs out of significant digits).

The way the precision scales with the magnitude of the value is the other important thing. When the exponent is 35, incrementing the first digit changes the value by 10^{35}, but when the exponent is 0, incrementing the first digit only changes the value by 1. This way you get angstrom accuracy when you're on the scale of angstroms, but not when you're on the scale of galaxies (when you really don't need it).

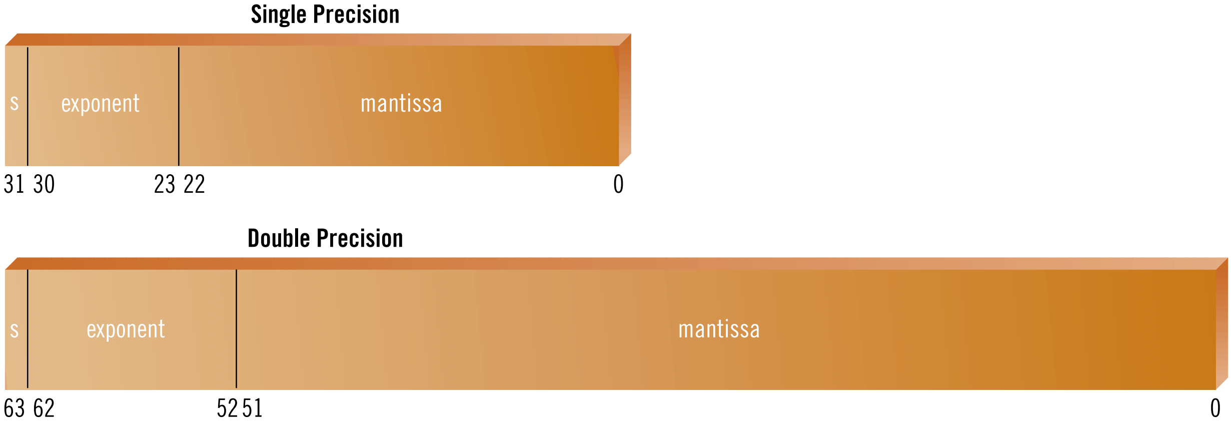
The IEEE floating-point standard specifies floating-point representations and how operations on those representations behave. The two IEEE floating-point formats we care about are "single" and "double" precision. They both share the same equation for conversion from the binary floating-point representation to the real number representation, and you'll recognize it as a binary form of scientific notation:

$$-1^{sign}\ x2^{exponent-bias}\ x1.mantissa \tag{1}$$

The only differences between the two formats are the widths of some of the named fields in Equation 1, so we'll go over each part of it in turn and point out the differences when they pop up.

We'll start on the right-hand side of Equation 1. The mantissa expression (the 1.mantissa part of the equation) is somewhat strange when you first look at it, but it becomes a little clearer when you realize that mantissa is a binary number. It's also important to realize that it is a normalized, binary number. "Normalized" for scientific numbers means the mantissa is always greater than or equal to 1 and less than 10 (in other words, a single, non-zero digit). Our previous example of scientific notation, 2.345 x 10^{35}, is normalized, while the same number represented as 234,500 x 10^{30} is not. When a binary number is normalized, it is shifted until the most significant bit is 1. Think about this for a second (I had to). If there are leading zeros in the binary number, we can represent them as part of the exponent, just as if there are leading 0 digits in our decimal scientific notation. And because the most significant bit is always 1, we can avoid storing it altogether and make it implicit in our representation. So, just like normalized decimal scientific notation keeps its mantissa between 1 and 10, the binary mantissa in a floating-point number is always greater than or equal to 1 and less than 2 (if we include the implicit leading 1).

Figure 1. IEEE Floating-Point Layouts



Next in line, the exponent expression shifts the binary point right or left based on a positive or negative exponent, respectively. This is exactly analogous to the base-10 decimal scientific notation, where the exponent shifts the decimal point right or left, inserting zeros as necessary. The exponent field is the key to the range advantage floating-point numbers have over fixed-point numbers of the same bit width. While a fixed-point number has an implicit binary point and all the bits are, in essence, a mantissa, a floating-point number reserves an exponent field to shift the binary point around (hence the term, "floating-point"). It's clear that 8 bits reserved for an exponent from a 32-bit word allows a range from about 2^{127} to 2^{-127}, while the best a fixed-point number could do would be a 32-bit range, for example 2^{32} to 0, 2^{16} to 2^{-16}, or 0 to 2^{-32}, but not all at the same time. However, there's no such thing as a free lunch; the 32-bit fixed-point number actually has better precision than the floating-point number, because the fixed-point number has 32 significant bits, while the floating-point number only has 24 significant bits left after the exponent is reserved.

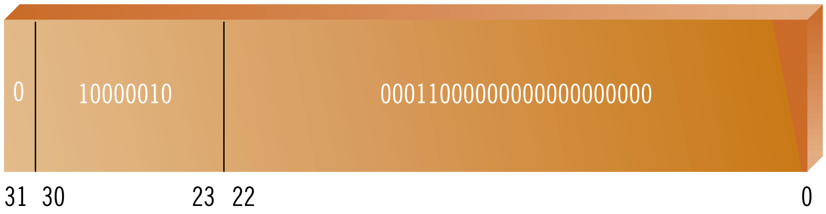
You'll notice the exponent expression is actually exponent - bias. The bias is a value set so that the actual bits of the exponent field are always positive. In other words, assuming the exponent is 8 bits and the bias is 127, if you want your unbiased exponent to be -126 you set your biased exponent bits to 1. Likewise, a biased exponent field value of 254 yields an unbiased exponent of 127. Exponent values of all 0s and all 1s are reserved for special numerical situations, like infinity and zero, but we don't have space to cover them.

Finally, the sign bit dictates whether the number is positive or negative. Unlike two's complement representations, floating-point numbers that differ only in sign also differ only in their sign bit. We'll discuss the implications of this further. Table 2 contains the field sizes for single and double precision IEEE floating-point values, and Figure 1 shows their layout, with the sign always at the most significant bit.

An Example

Let's run through an example by converting a decimal number into a single precision, binary, floating-point number. We'll use the number 8.75 because it's easy enough to do by hand, but it still shows the important points. First, we turn it into a binary fixed-point number, 1000.11, by figuring out which binary bit positions are 1 and 0. Remember, the bit positions to the right of the binary point go 2^{-1}, 2^{-2}, 2^{-3}, and so on. It should be clear that I chose .75 for the fractional part because it's $2^{-1} + 2^{-2}$, so it's easy to calculate. Next, we shift the binary point three positions to the left to normalize the number, giving us 1.00011 x 2^{3}. Finally, we bias this exponent by adding 127 for the single precision case, leaving us with 130 (or 10000010 bina-

Figure 2. 8.75 As a IEEE Single Precision Value



ry) for our biased exponent and 1.00011 for our mantissa. The leading 1 is implicit, as we've already discussed, and our number is positive, so the sign bit is 0. The final floating-point number's bit representation is shown in Figure 2.

Now that we're familiar with floating-point numbers and their representations, let's learn some tricks.

Conversions

I mentioned that on some processors the floating-point to integer conversions are pretty slow, and I wasn't exaggerating. On the Pentium, the instruction to store a float as an `int`, `FIST`, takes six cycles. Compare that to a multiply, which only takes three, and you see what I mean. Worse yet, the `FIST` instruction stalls the floating-point pipeline and both integer pipelines, so no other instructions can execute until the store is finished. However, there is an alternative, if we put some of the floating-point knowledge we've learned to use and build our own version of `FIST` using a normal floating-point addition.

In order to add two floating-point numbers together, the CPU needs to line up the binary points before doing the operation; it can't add the mantissas together until they're the same magnitude. This "lining up" basically amounts to a left shift of the smaller number's binary point by the difference in the two exponents. For example, if we want to add 2.345 x 10^{35} to 1.0 x 10^{32} in decimal scientific notation, we shift the smaller value's decimal point 3 places to the left until the numbers are the same magnitude, and do the calculation: 2.345 x 10^{35} + 0.001 x 10^{35} = 2.346 x 10^{35}. Binary floating-point works in the same way.

We can take advantage of this alignment shift to change the bit representation of a floating-point number until it's the same as an integer's bit representation, and then we can just read it like a normal integer. The key to understanding this is to realize that the integer value we want is actually in the floating-point bits, but it's been normalized, so it's shifted up to its leading 1 bit in the mantissa field. Take 8.75, as shown in Figure 2. The integer 8 part is the implicit 1 bit and the three leading 0s in the mantissa. The following 11 in the mantissa is .75 in binary fractional bits, just waiting to be turned into a fixed-point number.

Imagine what happens when we add a power-of-two floating-point number, like 2^8=256, to 8.75, as in Figure 3. In order to add the numbers, the CPU shifts the 8.75 binary point left by the difference in the exponents (8 - 3 = 5, in this example), and then completes the addition. The addition itself takes the implicit 1 bit in the 256 value and adds it to the newly aligned 8.75, and when the result is normalized again the implicit 1 from the 256 is still in the implicit 1-bit place, so the 8.75 stays shifted down. You can see it in the middle of the mantissa of the result in Figure 3. What happens if we add in 223, or the width of the mantissa? As you'd expect, the 8.75 mantissa is shifted down by 23 - 3 = 20, leaving just the 1,000 for the 8 (because we shifted .75 off the end of the single precision mantissa, the rounding mode will come into play, but let's assume we're truncating towards zero). If we read in the resulting single precision value as an integer and mask off the exponent and sign bit, we get the original 8.75 floating-point value converted to an integer 8!

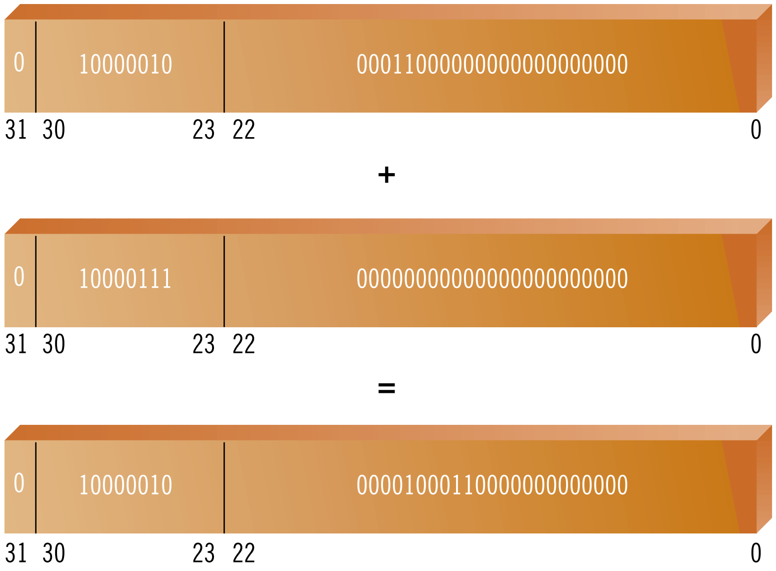
This trick works for positive numbers, but if you try to convert a negative number it will fail. You can see why by doing the aligned operation by hand. I find it easier to work by subtracting two positive numbers than by adding a positive and a negative. Instead of 2^{23} + (-8.75), I think of 2^{23} - 8.75. The single sign bit representation lends itself to this as well (using a piece of paper and a pen will really help you see this in action). So, when we do the aligned subtraction, the 8.75 subtracts from the large value's mantissa, and since that's all 0s (it's a power-of-two), the subtract borrows from the implicit 1 bit. This seems fine at first, and the mantissa is the correct value of -8.75 (shifted down), but the normalization step screws it up because now that we've borrowed from the implicit 1 bit, it's no longer the most significant bit, so the normalization shifts everything up by one and ruins our integer.

Table 2. Floating-Point Field Widths and Parameters

	Total Width	Sign	Mantissa	Exponent	Bias Value
Single	32 bits	1 bit	23 bits	8 bits	+127
Double	64 bits	1 bit	52 bits	11 bits	+1023

But wait, all is not lost. All we need is a single bit from which to borrow in the mantissa field of the big number so that the subtraction will leave the implicit 1 bit alone and our number will stay shifted. We can get this 1 bit simply by multiplying our large number by 1.5. 1.5 in binary is 1.1, and the first 1 becomes the implicit 1 bit, and the second becomes the most significant bit of the mantissa, ready to be used for borrowing. Now negative and positive numbers will stay shifted after their normalization. The masking for negative numbers amounts

Figure 3. Adding Single Precision Floating-Point Numbers



to filling in the top bits with 1 to complete the two's-complement integer representation.

The need to mask for positive and negative values is bad enough when you are only dealing with one or the other, but if you want to transparently deal with either, figuring out how to mask the upper bits can be slow. However, there's a trick for this as well. If you subtract the integer representation of our large, floating-point shift number (in other words, treat its bits like an integer instead of a float) from the integer representation of the number we just converted, it will remove all the high bits properly for both types of numbers, making the bits equal zero for positive values and filling them in with ones for negative values.

You'll notice that this technique applied to single precision values can only use a portion of the full 32 bits because the exponent and sign bits are in the way. Also, when we use the 1.5 trick we lose another bit to ensure both positive and negative numbers work. However, we can avoid the range problems and avoid masking as well by using a double precision number as our conversion temporary. If we add our number as a double (making sure we use the bigger shift value—2^{52} x 1.5 for integer truncation) and only read in the least significant 32 bits as an integer, we get a full 32 bits of precision and we don't need to mask, because the exponent and sign bits are way up in the second 32 bits of the double precision value.

In summary, we can control the shift amount by using the exponent of a large number added to the value we want to convert. We can shift all the way down to integer truncation, or we can shift part of the way down and preserve some fractional precision in fixed-point.

This seems like a lot of trouble, but on some processors with slow conversion functions, like the x86 family, it can make a difference. On the Pentium with `FIST` you have to wait for six cycles before you can execute any other instructions. But using the addition trick, you can insert three cycles worth of integer instructions between the add and the store. You can also control how many bits of fractional precision you keep, instead of always converting to an integer.

What's Your Sign?

Before I wrap this up, I'd like to throw out some other techniques to get you thinking.

The exponent bias is there for a reason: comparing. Because the exponents are always positive (and are in more significant bits than the mantissa), large numbers compare greater than small numbers even when the floating-point values are compared as normal integer bits. The sign bit throws a monkey wrench in this, but it works great for single-signed values. Of course, you can take the absolute value of a floating-point number by masking off the sign bit.

I've already hinted at the coolest trick—overlapping integer instructions while doing lengthy divides—but I haven't gone into detail on it. It will have to wait until next time, when we'll discuss this in depth.

Two people introduced me to the various tricks in this article and got me interested in the details of floating-point arithmetic. Terje Mathisen at Norsk Hydro first showed me the conversion trick, and Sean Barrett from Looking Glass Technologies made it work on negative numbers.

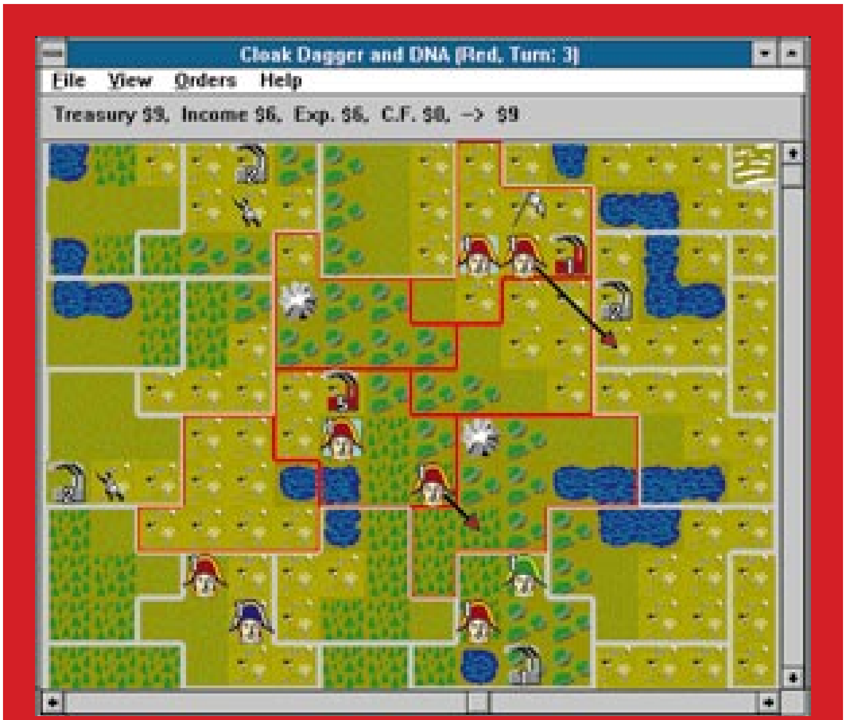
If you want to learn more about floating-point, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms* (Addison-Wesley, 1981) by D. Knuth is a good source for the theory. Most CPU programming manuals have fairly detailed descriptions as well. You can get Adobe Portable Document Format versions of the PowerPC manuals on http://www.mot.com. If you really want to understand floating-point and its implications for numerical programming, you'll want to pick up a numerical analysis textbook that deals with computers.

You also might want to look at the *Graphics Gems* series from AP Professional. The series covers a number of floating-point tricks like the ones discussed here. A good example calculates a quick and dirty square root by halving the exponent and looking up the first few bits of the mantissa in a table. Another takes advantage of the format to do quick absolute values and compares for clipping outcode generation. Once you understand the floating-point format, you can come up with all sorts of tricks of your own. ■

Chris Hecker tries to stay normalized, but he can be biased at checker@bix.com.

Back

Using Genetic Algorithms to Evolve Computer Opponents



```
Cloak Dagger and DNA (Red, Turn: 3)
Treasury $9, Income $6, Exp. $6, C.F. $0, -> $9
```

Using OID, you can program a simple genetic algorithm that will evolve into a reasonably intelligent computer opponent that doesn't resort to cheating to stay in the game.

I love a good strategy game. I've spent uncountable hours playing Civilization and Master of Orion. But I find I keep going back to my ancient Diplomacy (CGA yet!) and Risk programs with their klunky and awkward user interfaces because, as far as I can tell, they aren't "cheating" to play a good game.

To be fair, if computer strategy games had to play by the same rules, these games might still be in development or possibly never even started, and I certainly wouldn't want that. I'm sure I'll keep playing both games and others that cheat too. But whenever I think of buying new games, part of my evaluation includes trying to determine whether and how much they might have to cheat to be challenging.

One way to have a new strategy game that played fair was to write my own, so I set off to do just that. Hoping to have strong computer players without having to spend weeks or months writing them, I looked for a way to let my computer generate them with spare CPU cycles.

This article is a description of how I designed a game that could use genetic algorithms to generate some decent computer players. Although the computer players do not have access to all the information human players have, I have evolved some that can beat me in an evenly matched game—without cheating!

First, I'll outline the game, discussing the major design issues. In fact, there are many different rule options that can be selected when designing a scenario, so the rules I do mention will be the default rules in the basic game. Then, I'll show how the design supports computer players with a very small "genetic" program, along with some examples. Then, I'll show how easy it is to evolve these programs and how to test whether they really improve over time.

Currently, a very high interest in genetic algorithms exists, and it is getting easier to find articles and books on the subject with many examples and details. (If you have Web access, just do a search for genetic algorithms!) Unfortunately, very few genetic algorithm applications I've seen actually do anything interesting from a games perspective. Here, I'll try to concentrate on the design of "something interesting" to do with genetic algorithms and minimize the tutorial and theoretical aspects.

A Description of Cloak, Dagger, and DNA

The game is like a cross between Diplomacy and Stratego with a few mutations thrown in. The playing field is a map divided into areas, some of which contain factories. The factories are needed to support the armies that are used to capture more areas. Each turn, a factory supports one army or spy or puts one credit in the player's treasury that can be spent later. All players' armies and spies are moved simultaneously, so a turn consists of submitting a set of orders, not actually moving pieces.

The legality of moving pieces is based solely on the current position, so if you are legally able to construct an order, it will always be carried out. What you won't necessarily know until the next turn is how many of your armies survived in each area and who ended up controlling each area and any factories in it. When all players are ready, the game engine reads the human players' orders, generates the orders for the computer players, performs all moves, and resolves combat in areas where armies are owned by more than one player.

In the entire design, the combat resolution rules have seen the most revisions. My design goals for combat were: fair, simple, and repeatable. I have seen fair and repeatable with Diplomacy, which does those very well with no dice rolls. But if you've ever played noncomputer Diplomacy, you know how complicated resolving all the players' orders can get, as one set of orders very often prohibits another set from being carried out. The dependencies can be recursive, so it is not simple.

In my game, all legal orders are carried out, thus any pieces ordered built are placed on the map, and all pieces with orders to move are moved with no exceptions. Combat is then resolved in each area that contains armies belonging to more than one player (spies are not involved in combat). Combat does not necessarily result in only one player remaining in an area, in fact all four players can have one army each in a given area, and nothing will happen. However, armies in an area that contains other armies may only be ordered to leave that area if they move to another area already controlled by the owning player.

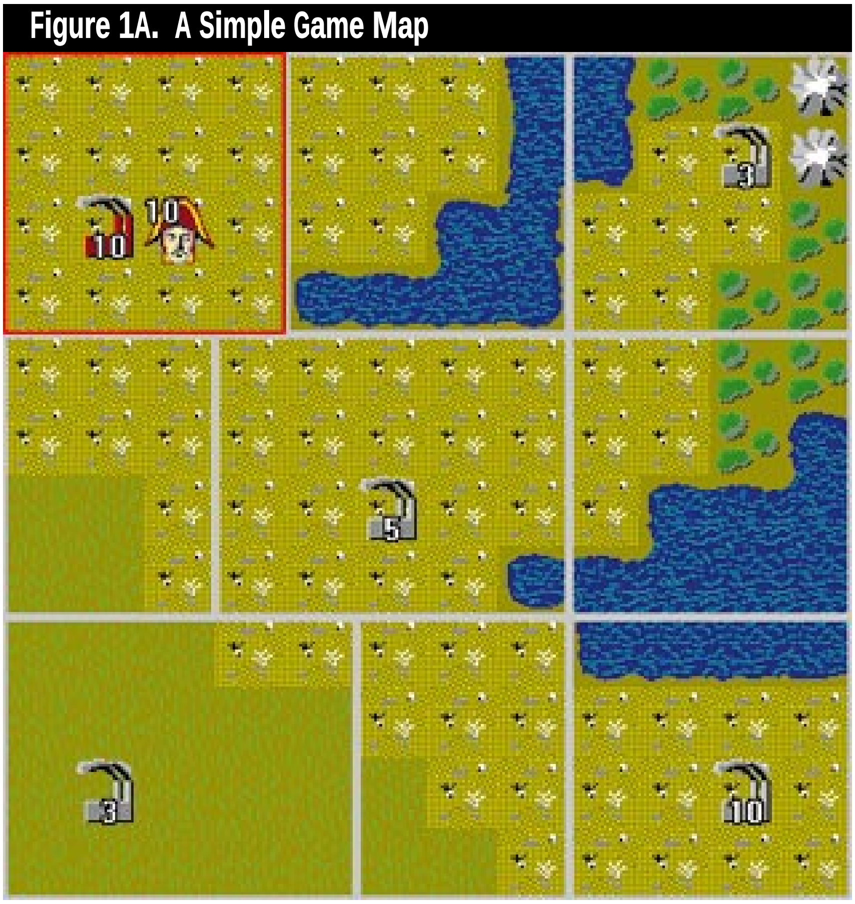
Each players' combat losses are computed separately, but in parallel, and are based on the number of armies the player has, the number of armies the single largest opposing player has, and who, if anyone, previously controlled the area being contested. A player in an area he or she controls loses one army for every three attacking armies in the largest opposing force. In an area he or she does not control, the player loses one army for every two armies in the largest opposing force. After combat, if only one player has surviving armies in an area, that player gains control of that area and any factories in it.

A player only knows about the other players' forces upon encountering them, which can be done intentionally by sending spies off into enemy territory. A player's map does show all the facto-

by Don O'Brien

If you love strategy games, but hate it when your computer has to cheat to stay in the game, learn about OID, the simple genetic algorithm that will evolve into a worthy opponent—and even beat you!

Figure 1A. A Simple Game Map



ries in the game, but not necessarily who controls them.

When a game is over, each player scores from 0 to 100, being the percentage of all factories in the game he or she controls. There are early win cases where one player can score all the 100 points. The actual scoring method is not important, as long as there is some relative fitness evaluation available to the genetic algorithm code.

Computer Player Genetic Codes

Inspired by Tom Ray's Tierra machine, I decided that the bits in the computer player genetic codes would be interpreted like machine code instructions. However, unlike the Tierra machine, a program that can make copies of itself isn't likely to be very similar to one that can play a strategy game. Intending to spend as little time as possible writing the initial opponents, I tried to design a fairly powerful virtual machine that would implement a simple computer opponent language. I call this the OID, which stands for Opponent Implementation Device.

It is also convenient that the suffix "oid" is used to mean an imperfect resemblance, since ultimately what we are doing when writing computer players is creating an imperfect resemblance of a live human opponent. I wanted the OID language to let me write a few simple seed programs with a small number of instructions that could play a game, even if poorly, and hoped that evolution would do the rest.

The OID language consists of a set of instructions that do not directly move individual pieces, but instead assign values to map areas and playing pieces. These values are used to decide which pieces, if any, move where. You can think of the map values as elevations and the pieces as marbles where their values are heights. With the right instructions, each marble on the map can add its height to the elevation of the area the marble is in. A single instruction can modify a register in each area on the map. For example, the instruction `Add R1 my Army` will add the number of the players armies, if any, to the R1 register in each map area.

Once a sequence of instructions has generated elevations on the map, a single move instruction can query all pieces individually to see if any want to "roll downhill" into adjacent areas. Any that do are given orders to move. Whenever a piece is given these orders, its height is used to reduce the elevation of the area it will leave and increase the elevation of the area it will enter before the next piece is queried.

For example, imagine that all areas with an enemy factory are assigned a low elevation, say, the number of factories as a negative number, and all areas with your armies are assigned a high elevation, say, the number of armies as a positive number. The `Move` instruction will then query all your armies to see if any are adjacent to areas with lower elevations, which will include empty areas next to areas with armies and, most importantly, adjacent enemy factories.

Building new pieces is similar, the Build instruction builds new pieces at factories at the lowest elevations first, adding the height of the piece each time, surplus treasury, and other thresholds permitting.

The following tiny OID program can play a game, although not very well. Yes, that is really all the code needed! If the other players do nothing each turn, this program will eventually take over most of the map in the basic game:

```
Sub R1 notmy Factory
Add R1 my Army
Build Army 1
Move Army 1
Halt
```

First, let's explore the architecture and environment. Then, we'll, step through this program line by line to see what it does in more detail.

The OID Architecture

The OID language has no loops or branching; all programs start at the beginning and run to the end. All OID programs have an (arbitrary) maximum of 25 instructions, including the Halt required at the end. If an instruction in the program does not specifically generate an order for a given piece, the default orders for that piece are to defend, which is the same as doing nothing.

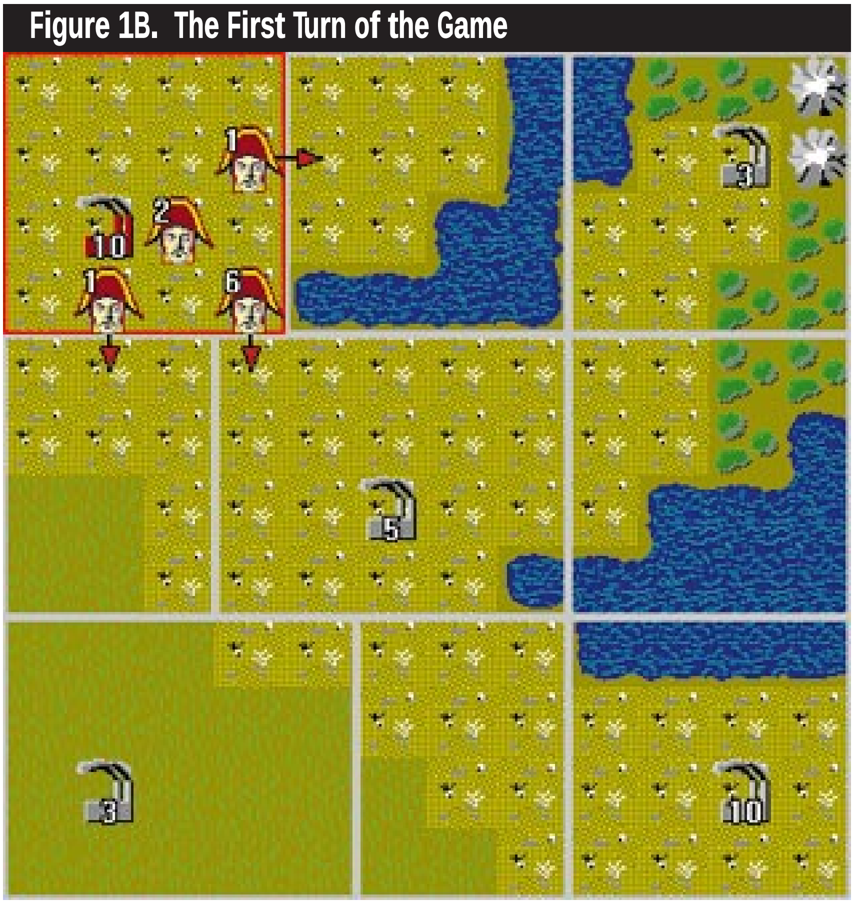
A program generated randomly in this language is guaranteed not to crash or cause any other kind of harm and to terminate in a finite time. This is necessary so that the genetic algorithms can “cross” and “mutate” programs without regard to the validity of the instructions or sequences of instructions. If a program does not generate any meaningful orders for a player, the pieces will just sit and defend until the end of the game or until they are destroyed by other players. The worst thing a program can do is disband all its armies and leave its factories undefended. A program only considers one turn in isolation. There is no provision to store information to be used in a later turn or keep track of what an opponent is doing over time.

The Area Registers

There are four signed integer registers for each area in the game: R0, R1, R2, and R3. These have independent values in each area and are processed (virtually) in parallel. For example, the instruction Add R3 my Factory inspects each area, and, if the area contains factories owned by the player running the OID program, each area has the number of factories in that area added to the area’s R3 register. The R3 registers in all other areas have zero added to them and remain unchanged.

At every program start, area register R0 is 1 for all areas, and R1, R2, and R3 are 0 for all areas. Most instructions that can change R1 are conditional on the value of R0 in the same area being greater than zero. Thus, for most instructions, if R0 is 0 or negative in a given area, an instruction that modifies R1 would be skipped for that area. In any other area where R0 is greater than zero, the instruction modifying R1 would be executed. R2 and R3 are general purpose area registers and have no special restrictions.

Figure 1B. The First Turn of the Game



OID Program Walk-Through

Now let’s take our sample program above, line by line. We will consider the game map in Figure 1A and assume we are running the OID program for the Red player. Figure 1B shows the first turn of a game after running the program. The pieces that have been ordered to move are shown with arrows indicating their moves.

Listing 1 shows the debug output produced by the OID when it runs a program, showing each instruction as it is encountered and showing a register set after any instruction that might modify that register. In this particular example, the numbers have been formatted in a three-by-three matrix to positionally match the areas they represent on the game map.

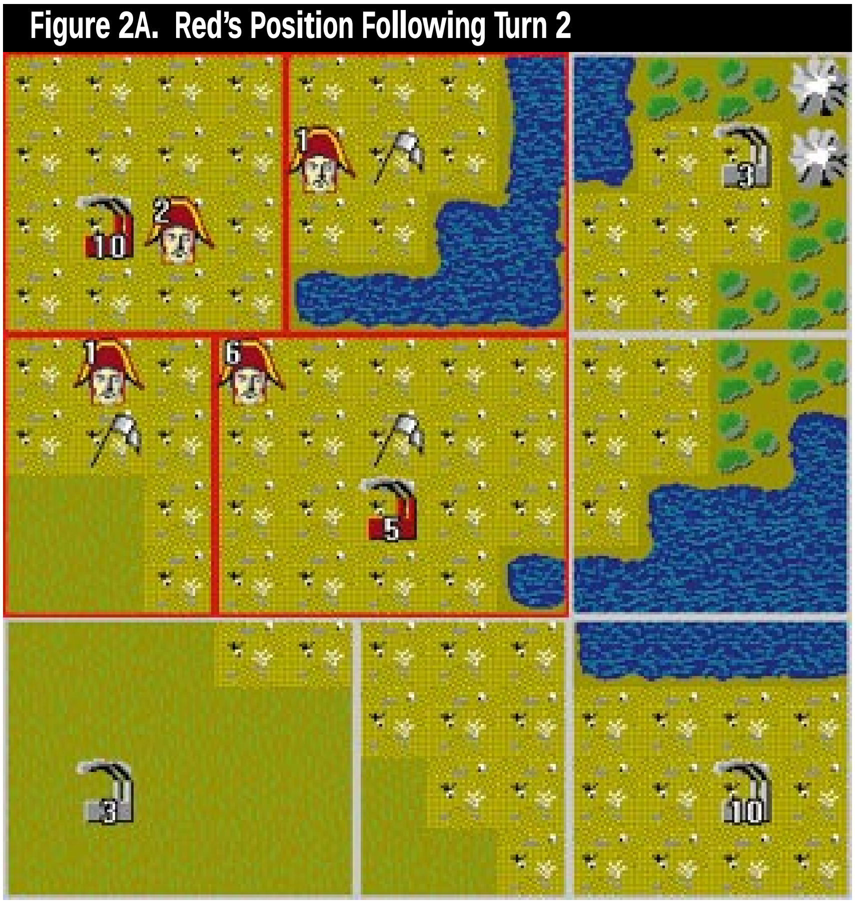
Since we know the program starts with all R0 registers set to 1 and R0 is not changed anywhere, all instructions in this program that operate on R1 registers will be performed. Also, all R1 registers will be initialized to 0.

The first line, Sub R1 notmy Factory, will subtract the number of enemy or unowned factories in each area from that

Listing 1. Debug Output

```
Turn 1
Sub R1 notmy Factory
R1 contains:
        0  0  -3
        0  -5 0
        -3 0  -10
Add R1 my Army
R1 contains:
        10 0  -3
        0  -5 0
        -3 0  -10
Build Army 1
    built=0
Move Army 1
    moved=8
R1 contains:
        2  1  -3
        1  1  0
        -3 0  -10
Halt
```

Figure 2A. Red's Position Following Turn 2



Listing 2. Turn 2

```
Turn 2
Sub R1 notmy Factory
R1 contains:
        0  0  -3
        0  0  0
        -3 0  -10
Add R1 my Army
R1 contains:
        2  1  -3
        1  6  0
        -3 0  -10
Build Army 1
    built=5
R1 contains:
        7  1  -3
        1  6  0
        -3 0  -10
Move Army 1
    moved=7
R1 contains:
        5  2  -3
        2  1  1
        1  0  -10
Halt
```

area's R1 register. After the Sub R1 notmy Factory instruction, you can see the negative values of all non-Red owned factories in the game. The factory in the upper left is already owned by the Red player and does not qualify as a notmy Factory, leaving the upper left R1 register unchanged.

After the Add R1 my Army instruction, the upper left R1 register contains 10, since Red has 10 armies in upper left area. No other registers are changed.

The Build instruction needs a little more explanation. There are some other specific purpose registers that are used by the Build and Disband instructions, relating to the players' current cash flow and treasury situation. Unfortunately, they are one of the weaker aspects of this design, as they don't provide much flexibility to make decisions about the relative proportions of armies and spies or to change these proportions under different circumstances. They, like the area registers, are initialized to 0 at program start. If we don't change them, the Build instruction will default to building exactly enough armies to make the Army and Factory totals the same.

The Build Army 1 instruction does nothing, since Red currently has 10 armies and 10 factories.

The Build and Move instructions do not have a register field and always use the R1 register. The Move instruction essentially queries all unordered armies one by one to see if any want to move. More specifically, it sorts all areas with unordered armies by elevation and, starting with the highest elevations and working down, determines whether an army should move.

First it removes the armies height from that area's R1 register and looks for an adjacent area with the lowest elevation. If the lowest adjacent area is lower that the origin area, the army is given orders to move to the adjacent area, and the adjacent area's R1 register is increased by the army's height. The adjacent area is immediately repositioned in the Move's sort list if necessary, and, if there are still unordered armies in the origin area, the next army is considered. If an army is not moved, its height is added back to the R1 register of its area of origin, and the Move instruction proceeds to the next area in the list.

Now, the Move Army 1 instruction moves eight armies, and, as you can see in the R1 matrix following Move Army 1 in Listing 1, it manages to even out the elevations of the three adjacent areas. The upper left is still one unit higher than its neighbors, but since an army subtracts its own height before considering a move, the last army that was considered for moving did not see a lower elevation.

Following with turn 2, seen in Figure 2A, Red has not encountered any other players, but has gained the five factories in the center of the map. Skipping down to the Build Army 1 instruction, we see that it will now come into play, as Red has more factories than armies. Listing 2 shows the OID output for turn 2, and Figure 2B shows the map after the run. The Build instruction had to build five armies and choose between two factories. As you can see in Listing 2, before the Build instruction, the elevation of the upper left factory was 2, and the center factory was 6. The first four armies were built at the lowest elevation factory in the upper left, and for the fifth factory a choice had to be made because the elevations were equal.

When given such a choice, my implementation always chooses the area with the lowest internal index number, which in this case is the upper left factory. The advantages are reproducible for debugging and fitness testing for the genetic selection, since in both cases a given game between OID-run players will always produce exactly the same result. The disadvantages are that on a rotationally symmetric board, OID players will not play symmetrically, and when playing against a human, OID players are very predictable, once you get to know them and the map.

As you can see in Figure 2B, this Red program does not protect the center factory very well. In fact, if the same OID program was playing for a different player from a similar start in the lower right corner, that player would take the center factory from Red on turn 3.

Applying the Genetic Algorithms

Now we have a design that may support genetic algorithm evolution. After a quick review of the genetic algorithm technique and some hand-constructed examples, I'll try to evolve some new players with the OID that was constructed with the design outlined here.

The genetic algorithm technique for solving a problem is, simply stated:

1. Start with a pool of candidate solutions.
2. Evaluate the fitness of each solution.
3. Replace some solutions with combinations of others.
4. Possibly make random modifications to the newly created solutions.
5. Go to step 2.

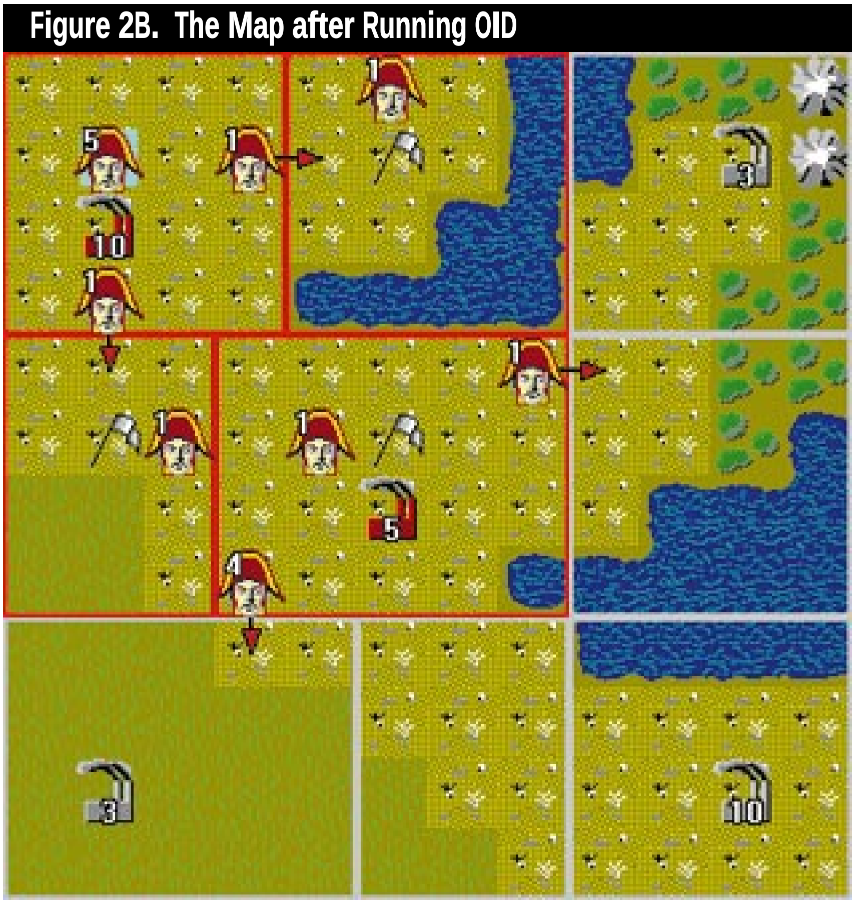
With the proper fitness function for step 2, selection criteria for step 3, and mutation rate in step 4, your pool of solutions should improve as this process iterates.

Generating an initial pool of OID programs is, by design, reasonably simple. They don't have to be good, but they have to play well enough that if we put them in a game they would do better than just sitting in their initial positions. We have seen this program play well enough for this.

In this implementation, we can only evaluate relative fitnesses of the OID programs. To do that, they must play against each other. The scoring system I use can be thought of as food for the programs. The programs consume a set amount of food to enter a tournament, and the winners are rewarded with more food. After a number of games, the food will be roughly proportional to the relative fitness of a program, and when a program's food supply goes below zero, it becomes a candidate for replacement.

The game allows up to four players, letting the OID evolution engine evaluate four players relative to each other at one time. The map in the basic game does not necessarily contain starting

Figure 2B. The Map after Running OID



positions of equal fairness, so four games are always played as a tournament, with the players rotating seats after each game. I could have used all 24 possible combinations of four players in four seats, but it seemed like overkill. After each tournament, each participating program receives the average of its scores from the four games in food points.

I wrote the Evolution Engine to allow the user to control most things that seemed interesting, including:

- The maximum food a pool member can store
- How much food it costs to enter a tournament
- How much food it costs when chosen as a parent
- How much food a newly created pool member starts with
- The mutation rate
- How parents are chosen
- How pool members are chosen for each tournament.

There are many possible combinations of parameters and techniques, some more or less efficient than others, and some that do not produce useful evolution at all. We could write yet another genetic algorithm to evolve an efficient parameter set for this one. The description that follows is an extremely over-simplified version of what the OID does, without worrying too much about parameter settings.

Table 1. First Tournament

Gene	0	1	2	3	
Red	27	0	0	24	12
Yellow	56	1	18	0	18
Green	100	4	18	21	35
Blue	59	0	14	52	31
Score	60	1	12	24	

A (Simplified) Tournament Walk-through

Four pool members are chosen to play a tournament. I wrote the following four programs by hand for this example of player evolution:

```
// Gene 0 (from above)
Sub R1 notmy Factory
Add R1 my Army
Build Army 1
Move Army 1
Halt

// Gene 1
Add R1 my Army
Mul R1 5
Build Army 2
Move Army 1
Halt

// Gene 2
Distf R1
Distf R1
Add R1 my Army
Move Army 5
Build Army 1
Halt

// Gene 3
Sub R1 notmy Factory
Avg R1 1
Add R1 my Army
Move Army 1
Build Army 1
Halt
```

After running a tournament with these programs, the results shown in Table 1 were produced. This matrix shows the gene programs in the columns and the seating positions in the rows. As you can see in the first column, gene 0's average score from the four games was 60 points. The results from the four individual game scores are placed diagonally in the matrix, so the map positions can also be scored in the rows. I use this to help me tune the starting positions as much as possible for fairness. (Because of rounding, scores from any given game or tournament will often add up to less than 100. So far, no computer players have evolved sufficiently to complain.)

In this particular tournament, Red seems to be a weaker position than Green, but this minor difference could easily change with different players. Also, with these numbers, we might do some statistical comparison between the seating results in the right most column and the gene score results across the bottom.

For example, if the Red position won every game played, the tournament scores for each gene would be 25, and we wouldn't have learned anything except to play Red if you want to win. You could play around a little with statistical analysis in an attempt to evaluate the significance of any tournament. However, because we rotate the seating positions, if a game is not fair, it will tend to have only minimal effect on the relative fitness comparisons of the genes, since they will tend to get very similar scores.

Survival of the Win(est)

Let's assume that gene 1 needed to win more than 1 food point to survive and is replaced. The OID implements gene crossing by taking a random number of instructions from the beginning of one program and concatenating them to a random number of instructions from the end of another program. The random numbers are chosen, so the resulting program never exceeds the 25 instruction limit.

If we choose the new parents from the two highest scoring genes, 0 and 3, and cross them without mutation, we might get the following result:

```
Sub R1 notmy Factory    // 0
Add R1 my Army          // 0
Build Army 1            // 0
Avg R1 1                // 3
Add R1 my Army          // 3
Move Army 1             // 3
Build Army 1            // 3
Halt                    // 3
```

There is now a redundant `Build` instruction of which, in this particular program, the second will never build anything. However, the interesting thing is that some instructions have been inserted between a `Build` and `Move` instruction. `Build` and `Move` are using different elevation maps to make their decisions. None of the original four programs does this.

Let's run another tournament with the new gene 1. The results are shown in Table 2. There were no miracles here, but the new gene 1 is a definite improvement over the old.

Does It Really Work?

I loaded the original four gene programs into the OID as the first of 50 pool members. I filled the remaining 46 pool entries with random crosses and mutations of the original four. I locked the original four so they would not be deleted if they ran out of food; they would just maintain a negative food supply. Then I ran 300 tournaments or 1,200 total games on a map that has 24 areas. This took about 30 minutes on my 486DX2-66. I scanned the resulting pool for a gene with a high food score and found the following program for gene 48:

```
Sub R1 notmy Factory
DistF R1
Mul R1 5
Add R1 my Army
Move Army 1
Build Army 1
Add R1 my Army
Build Army 1
Move Army 1
Halt
```

I noticed that all four original programs showed a food supply in the negative hundreds. Just to be sure, I set up a tournament with this new program and three of the original genes: 0, 2, and 3, just to see what would happen. Gene 48 scored 100 in all four games!

I played a game as a human with the original genes 0, 2, and 3 as the opponents, and they were pretty easy to beat. I won after just a few turns. Then, I played another with gene 48 running the other three opponents, and I had to work a little bit to win. In fact, a couple of turns took quite a bit of thinking, and I came close to losing factories more than once. For 30 minutes of evolution, that's not too bad.

Seeing the Enemy

Between the other users of the OID program and me, we've run a little more than a million tournaments. Many of the newly evolved players can consistently beat me—that is, until I learn what they are doing. The game interface allows human players to run an OID program as an advisor and accept, ignore, or modify the orders advised. Because I can learn techniques for playing my game from them, I get a little better and can eventually beat the better ones more often than not.

With a big enough map, say 50 or more areas, which is a little more than a Risk map and a little less than Diplomacy, I always seem to make a mistake somewhere, and the enemy breaks through the front lines. This almost always leads to unrecoverable disaster—even when I use spies, and no OID program has evolved to use spies very well.

Table 2. Second Tournament

Gene	0	1	2	3	
Red	27	25	16	24	23
Yellow	39	28	18	28	28
Green	29	21	18	21	22
Blue	33	25	14	25	24
Score	32	24	16	24	

All in all, I've got some tough opponents, which I didn't have to spend weeks or months writing. I don't cheat, and occasionally I even feel good about it when I lose because, after all, I created them, even if only indirectly. ■

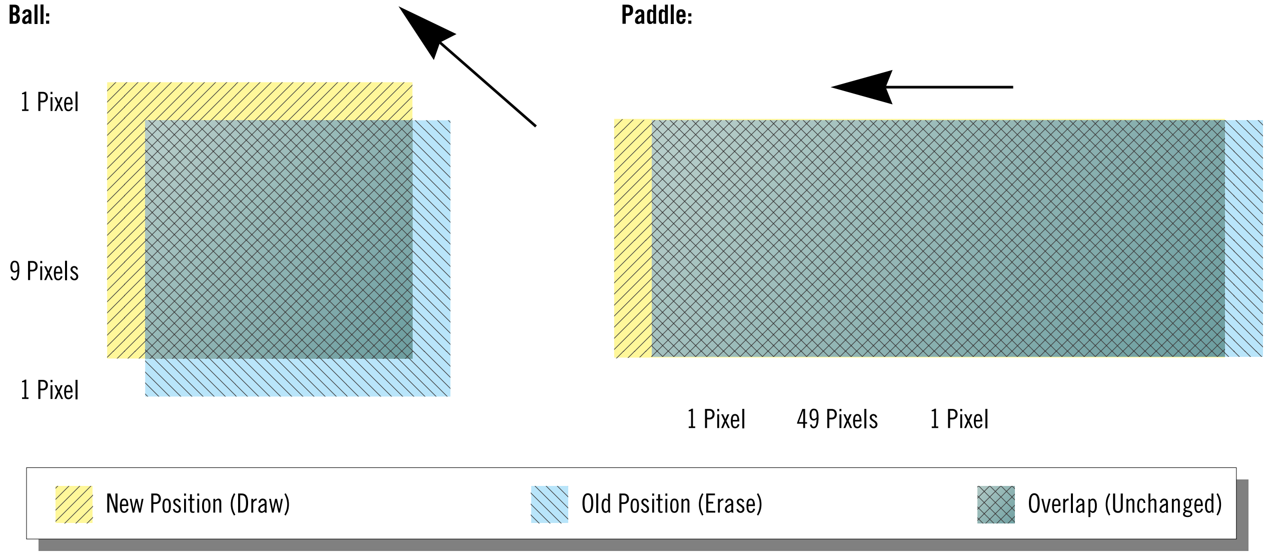
Don O'Brien is on a quest for ways to make a living and have a life at the same time after many long years of PCB CAD system design. You can reach him at 71702.2255@compuserve.com.

Tom Ray's Tierra program is described in "An Approach to the Synthesis of Life" (*Artificial Life II*, Addison-Wesley 1992, pp. 371-408).

Back

XSplat Revisited

Figure 1. Ball and Paddle Motion



Witty subject headings and an engaging opening line stump hundreds of technical writers every month. The technical part of an article like this comes easily; a programmer should be able to crank out several pages describing a piece of code if he or she has thought it through carefully. It's repackaging all that technical drivel into human speech that makes the writing tough.

This month, I hit a wall trying to segue from my last XSplat article to this one. I wanted to talk about how I've presented enough XSplat in the last few installments of this series to move away from examining multiple implementations of the platform-specific stuff and toward demonstrating the real gold mine of cross-platform development—which is, of course, the ability to tune code independently of the target platform.

My girlfriend suggested I start the article by telling a few jokes. Relax the crowd a bit. Your basic toastmaster type of stuff. I thought about trying the "three strings walk into a bar..." bit, but I think I'll just skip the preamble and cut to the chase.

But before I begin, I should remind you that the code for this article (and for others in this magazine) is available at the *Game Developer* ftp site, which is in ftp://ftp.mfi.com in the /pub/gamedev/src directory, or on CompuServe in the Game Developer Library of SDFORUM.

XSplat Plumbing

We need to look at two things related to the framework—they'll fill in a couple of gaps before we get started on this month's antics.

I promised to provide a SwapRect function to augment the SwapBuffer function, which is all we've had until now. On both Macintosh and Windows, this is just a general case of the SwapBuffer code, so I'll leave you to guess at its implementation or sneak a peek at the *Game Developer* ftp site. The function prototype looks like this:

```
void COffscreenBuffer::SwapRect(int
  Left, int Top, int Right, int Bottom)
  const;
```

Here's a hint. You can now implement COffscreenBuffer::SwapBuffer as an inline call to COffscreenBuffer::SwapRect like this:

```
inline void COffscreenBuffer::Swap
  Buffer(void) const
{
  SwapRect(0, 0, Width, Height);
}
```

SwapRect is bottom-right exclusive, meaning that column Right and row Bottom don't get copied. To swap a single pixel (a 1-by-1 rectangle), call SwapRect(x, y, x+1, y+1).

We're also going to need a function to query the system clock. Macintosh and Windows both provide straightforward ways to get this information, so we can just do some simple conditional compilation. Here's the function we'll use for now:

```
inline long unsigned GetMillisecond
  Time(void)
{
#if defined(_WINDOWS)

     return timeGetTime();

#elif defined(_MACINTOSH)

   return (long unsigned)TickCount() *
  1000 / 60;

#endif
}
```

On Windows, timeGetTime returns the current value of the multimedia timer, in milliseconds. Under Windows NT, this function can have latencies around 5 to 10 milliseconds, depending on the hardware and NT version. The solution to this latency problem is to use QueryPerformanceCounter, but I'm not going to do that right now. Under Windows 95, timeGetTime is millisecond accurate. On the Macintosh, TickCount returns the number of ticks since the computer started, with one tick being 1/60 of a second, so it's accurate only to about 17 milliseconds.

Now that we've added these platform-specific implementations to our XSplat repertoire, we can tackle the platform-independent problem of speeding up graphics. I'll shut up about XSplat for the rest of this article.

Shoveling Dirt

Watching the sprite move in the sample program made me feel like I was watching the arrival of the next Ice Age in real time. There aren't many games that could be slower: if you played it for too long, spiders may have started spinning webs across your arms. I promised to make it faster, though, and that's what I'm going to do.

Jon Blossom

Editor's Note: In Part II of this series, which appeared in the December 1995/January 1996 issue, Jon Blossom used a simple game format to demonstrate his approach to cross-platform programming that played similarly to the classic Atari video game "Breakout." Jon and the Editors of Game Developer *want to remind our readers that "Breakout" is a protected trademark and copyrighted work of Atari Corp. (copyright 1983-1996 Atari Corp., all rights reserved). Unauthorized use of others' code or audiovisual elements may constitute a violation of the owner's intellectual property rights. Use of the "Breakout" name in Jon's programming exercise was not intended, and should not be interpreted, as an endorsement by* Game Developer *of any unauthorized use of Atari's "Breakout" trademark or copyrighted game. We regret any misunderstanding.*

The code Jon has written for this series of articles is now available on the Game Developer ftp site, ftp://ftp.mfi.com/pub/gamedev/src/. It includes the various techniques and utilities discussed in the series, and a new game to demonstrate their use.

Speeding up a game that ignores the basic code vocabulary of game programming is a no-brainer. Ours is basically a sprite-based game, and no software-sprite-based game should be without dirty rectangles. I'm guessing that anyone reading this magazine has heard of dirty rectangles, but here's a quick review: a dirty rectangle algorithm lets you take advantage of visual similarities between frames by updating only the areas of the screen that have changed.

In one implementation of a dirty rectangle algorithm, an application stores an image of the game in an offscreen buffer. When an object in the game moves, it erases the object in its old position in the buffer and draws it again in the new position, recording the coordinates of the areas affected by the change. To update the view, the game only has to `blt` the areas that have changed to the screen.

In my sample game, I redrew and copied the entire game to the screen every frame, even though only a small portion of the rendered game changes when the sprites move. Look at some numbers, and you'll see why dirty rectangles are great. In the entire 500-by-350-pixel window, there are 175,000 pixels, but the only things that change position in an average frame are small sprites. The ball is 10-by-10 and moves diagonally one pixel in either direction, so it affects an 11-by-11—or 121—pixel area. The player's piece is 50-by-10 and will move one pixel horizontally, so it affects an area that's 51 by 10, or 510 pixels.

In other words, the game updates 100% of the frame even though only 0.32% of the frame changes. What idiot would do that? We can cut away those 749,369 extra pixels by shoveling only the dirty areas to the screen.

In fact, every pixel in the ball is the same color, so there's no point in redrawing the whole thing. All we have to do is erase the trailing edge and draw in the leading edge, so instead of affecting 121 pixels, the moving ball actually changes only 38 pixels. The same goes for the player's piece: instead of 510 pixels, it actually changes only 10 pixels in front of it and 10 pixels behind it, as shown in Figure 1. Only 0.03% of the image changes every frame.

There's a tenfold difference between 0.03% and 0.32%, but I'm going to implement the 0.32% version anyway. The truth is, it's not going to make much of a difference. Video bandwidth, not main memory bandwidth, is generally a larger bottleneck, and we're still going to swap in the complete 11-by-11 and 51-by-10 areas. If the sprites were larger, it would make sense to spend more effort optimizing the `blt`s, but this will be plenty fast. Besides, my computer freaked out on me, and I had to kill off my compiler before I could implement the 0.03% algorithm.

> A dirty rectangle algorithm lets you take advantage of visual similarities between frames by updating only the areas of the screen that have changed.

Timing Valves

The source code for the 0.32% dirty rectangle rendering algorithm appears in Listing 1. But wait! There's more! If you implement that algorithm and try to play the game, you'll find that the Ice Age has become the Warp Age. Now we're in game programmer's heaven—that mythical land where our game does everything we want it to do, and it's actually *too fast*.

That's wonderful. Slowing a game down is infinitely easier than speeding it up. We just use the timing functions I introduced earlier to put two valves on the game that control the speed of the paddle and the speed of the ball. We'll need two new members of `CPaddleGameWindow`, called `NextBallTime` and `NextPaddleTime`, which tell the game when next to animate the two pieces, and two others, called `BallSpeed` and `PaddleSpeed`, which tell the game how to compute the `NextBallTime` and the `NextPaddleTime`.

During `Idle`, the game checks the current time. If it's later than `NextBallTime`, it calculates the motion of the ball and advances `NextBallTime` to be the current time plus `BallSpeed`. If it's later than `NextPaddleTime`, it calculates the motion of the paddle and sets `NextPaddleTime` to be the current time plus `PaddleSpeed`. We control the speed of the game by adjusting the two speed variables. The higher the speed value, the slower that piece of the game will move.

Listing 1 shows the heart of the modified `CPaddleGameWindow::Idle` code. To help experiment with it, I also implemented code in `KeyDown`, so you can control the ball speed using the + and - keys. You'll find that code on the ftp site if you're interested.

Full Throttle

The coolest thing about timing valves is that you can open them up as the game progresses, pushing the game faster and faster as the player completes more and more levels. To determine whether a player has completed a level, `CPaddleGameWindow` keeps track of how many blocks remain using the `BlockCount` member, decremented by `CPaddleGameWindow::HitBlock` when a wall is destroyed. If `BlockCount` reaches zero, `CPaddleGameWindow::InitGame` will reset for the new level.

Listing 1. Mostly Optimized XSplat

```
void CPaddleGameWindow::Idle(void)
{
    // Don´t do anything while backgrounded
    if (!IsActiveFlag)
        return;

    // Control the frame rate
    long unsigned CurrentTime = GetMillisecondTime();
    if (CurrentTime < NextBallTime &&
         CurrentTime < NextPaddleTime)
        return;

    //---------------------------------------------------------------
    // Prepare the buffer

    COffscreenBuffer* pBuffer = GetOffscreenBuffer();
    if (!pBuffer)
        return;

    pBuffer->Lock();

    if (CurrentTime >= NextBallTime)
    {
        NextBallTime = CurrentTime + BallSpeed;

        // Erase the ball
    int BallDirtyLeft = BallX;
    int BallDirtyRight = BallX + kBallSize;
    int BallDirtyTop = BallY;
    int BallDirtyBottom = BallY + kBallSize;
    FillRectangle(pBuffer,
        BallX, BallY, kBallSize, kBallSize,
        kColorGameBackground);

        //-----------------------------------------------------------
        // Move the ball and calculate a bounce off any walls

        // ***** CODE OMITTED: SAME AS LAST MONTH *****
        // *****     AVAILABLE ON THE FTP SITE    *****

        // Enlarge the ball´s dirty rect
        if (BallDirtyLeft > BallX) BallDirtyLeft = BallX;
        if (BallDirtyRight < BallX + kBallSize)
            BallDirtyRight = BallX + kBallSize;

        if (BallDirtyTop > BallY) BallDirtyTop = BallY;
        if (BallDirtyBottom < BallY + kBallSize)
            BallDirtyBottom = BallY + kBallSize;

        // Draw the ball in its new position
    FillRectangle(pBuffer,
        BallX, BallY, kBallSize, kBallSize,
        kColorBall);

    pBuffer->SwapRect(BallDirtyLeft, BallDirtyTop,
        BallDirtyRight, BallDirtyBottom);

        //-----------------------------------------------------------
        // Perform any velocity changes due to bounces

        if (BounceX) BallXSpeed = -BallXSpeed;
        if (BounceY) BallYSpeed = -BallYSpeed;
    }

    //---------------------------------------------------------------
    // Follow the ball with the paddle center when in demo mode,
    // allow the user to control the paddle in play mode

    if (CurrentTime >= NextPaddleTime)
    {
        // Erase the paddle
    int PaddleDirtyLeft = PaddleX - kPaddleWidth/2;
    int PaddleDirtyRight = PaddleX + kPaddleWidth/2;
    FillRectangle(pBuffer,
        PaddleX - kPaddleWidth/2, kPlayAreaBottom - kPaddleHeight,
        kPaddleWidth, kPaddleHeight,
        kColorGameBackground);

        NextPaddleTime = CurrentTime + PaddleSpeed;

        if (IsDemoMode)
        {
            if (PaddleX < BallX + kBallSize/2) ++PaddleX;
            else if (PaddleX > BallX + kBallSize/2) --PaddleX;
        }
        else
        {
            PaddleX += PaddleXSpeed;
        }

        // Make sure the paddle doesn´t move out of the play area!
        if (PaddleX < kPlayAreaLeft + kPaddleWidth/2)
            PaddleX = kPlayAreaLeft + kPaddleWidth/2;
        else if (PaddleX > kPlayAreaRight - kPaddleWidth/2)
            PaddleX = kPlayAreaRight - kPaddleWidth/2;

        // Enlarge the paddle´s dirty rect
        if (PaddleX - kPaddleWidth/2 < PaddleDirtyLeft)
            PaddleDirtyLeft = PaddleX - kPaddleWidth/2;
        if (PaddleX + kPaddleHeight/2 > PaddleDirtyRight)
            PaddleDirtyRight = PaddleX + kPaddleWidth/2;

        // Draw the paddle in its new position
    FillRectangle(pBuffer,
        PaddleX - kPaddleWidth/2, kPlayAreaBottom - kPaddleHeight,
        kPaddleWidth, kPaddleHeight,
        kColorPaddle);

    pBuffer->SwapRect(PaddleDirtyLeft,
        kPlayAreaBottom - kPaddleHeight,
        PaddleDirtyRight, kPlayAreaBottom);
    }

    pBuffer->Unlock();
}
```

Every five complete levels, we'll increment the number of hits required to destroy each block, and for every level within those five, we'll increase the ball speed by seven frames per second. If we include a CurrentLevel member variable, the number of hits required to destroy a single block will be CurrentLevel/5 + 1, and BallSpeed will be 1000 / (30 + 7 x (CurrentLevel % 5)). Adding a few entries to the kColorBlock block array supplies colors for blocks requiring more than one hit to destroy.

Listing 2 shows the new version of CPaddleGameWindow::InitGame that implements this behavior for us. Only one problem (for now): if you're playing the game on a system with a low-resolution timer, the timing valves don't work as smoothly as they should. For instance, if

Listing 2. XSplat Level Initialization

```
void CPaddleGameWindow::InitGame(void)
{
    // Start the paddle in the middle of the play area, not moving
    PaddleX = (kPlayAreaRight - kPlayAreaLeft)/2 + kPlayAreaLeft;
    PaddleXSpeed = 0;
    // Start the ball just above the paddle
    BallX = (kPlayAreaRight - kPlayAreaLeft)/2 +
        kPlayAreaLeft - kBallSize/2;
    BallY = kPlayAreaBottom - kBallSize - kPaddleHeight;
    // Move the ball towards the upper-left
    // TODO: Randomize initial ball velocity
    BallXSpeed = -1;
    BallYSpeed = -1;
    // Start Slowly - ball at 30 frames per second, increasing 7 fps every level
    BallSpeed = 1000 / (30 + 7 * (CurrentLevel % 5));
    NextBallTime = GetMillisecondTime() + BallSpeed;
    // Initialize the game field
    BlockCount = kWallWidthBlocks * kWallHeightBlocks;
    int HitCount = CurrentLevel / 5 + 1;
    int Count;
    for (Count = 0; Count < BrickCount; ++Count)
        GameField[Count] = HitCount;
    // Draw the complete initial game state
    COffscreenBuffer *pBuffer = GetOffscreenBuffer();
    if (pBuffer)
    {
        pBuffer->Lock();
        DrawCompleteGameState();
        pBuffer->Unlock();
    }
}
```

you're playing on a Macintosh, `GetMillisecondTime` returns numbers in increments of 17. The six-millisecond changes in ball speed won't be noticeable for three levels, when they finally exceed the resolution of the timer. At that point, you'll see one big jump in the speed of the ball. Timer resolution can be a huge hassle for programs like this, but it's a topic for another day.

Speeding it up and adding increasing levels of difficulty pushes our paddle game play forward a long way. Players can still drop the ball with no penalty, which means they don't actually have to do anything, but we can fix that later.

A Frayed Knot

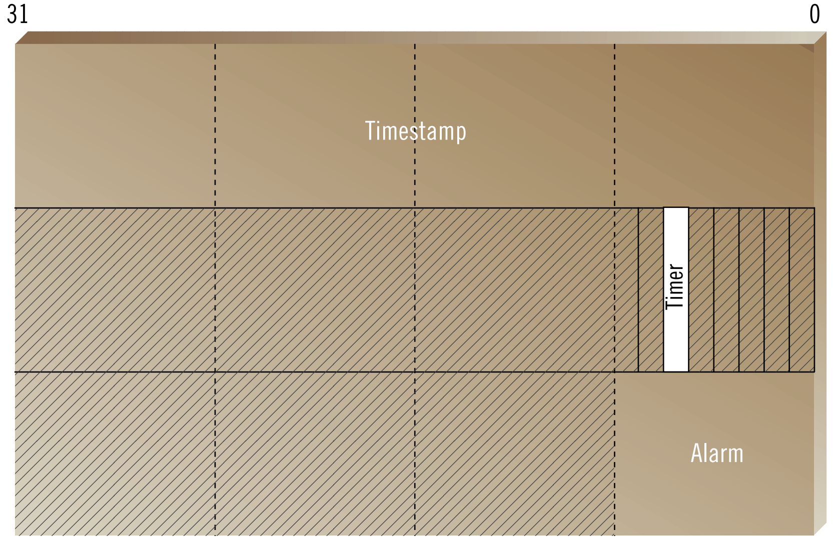
As for the three strings and the bigoted bartender, I can't even remember the whole joke, I can only remember how bad it is. ■

You can reach Jon Blossom via e-mail at blossom@slip.net or through Game Developer *magazine.*

Back

Organizing User Input, Part II: Mouse, Timer, and User-Defined Events

Figure 1. Timer Event Structure



Welcome back! Last time, if you recall, we discussed the input queue concept, the priority queue implementation of the input queue manager, and how to capture and enqueue keyboard events. In this concluding article, we're going to look at the other types of events the input queue manager can handle: those events generated by the mouse and timer as well as events defined and posted by the user. As an added bonus, we're also going to talk about drawing the mouse cursor ourselves, in any video mode, so we don't have to worry about whether or not the mouse device driver supports that mode.

Finally, at the end of the article, I'll introduce a simple example game that's both gratuitously violent and a fine demonstration of the input queue manager's capabilities.

Timer Events

The input queue manager abstracts the interface to the system timer through countdown alarms. When an alarm is initialized, it's given a starting count. With each clock tick, the alarm's count is decremented. When the alarm's count reaches zero, a `timer_alarm` event posts to the input queue, shown in Figure 1.

Alarms come in two flavors: one-shot and continuous. A one-shot alarm is disabled when its starting count reaches zero. A continuous alarm reenables itself with the same starting count as soon as it posts a `timer_alarm` event. The input queue manager provides up to 16 alarms, numbered 0 through 15, which are enabled using the `INPQ_set_alarm()` function. This routine accepts an alarm number and the starting count. It also takes a Boolean value indicating whether or not one-shot operation is desired.

Before the alarm abstraction is feasible, we need to provide a timer with a resolution that is fine enough to be useful. Let's face it, the PC's standard resolution of 54.9 ms just doesn't cut it. To much can happen in 54.9 ms, especially with today's computers. Even the resolution that I settled on, 1 ms, is rather coarse; but if it were any faster, we'd spend altogether too much time in the timer interrupt handler to get any other work done.

Without going into too much gory detail, the programmable interval timer (PIT) chip has three timer channels, each dedicated to a specific task. Channel 0 handles the system clock, channel 1 takes care of DMA memory refresh, and channel 2 controls the PC speaker.

Of the three channels, we are going to choose to share the channel controlling the system timer: channel 0. We certainly wouldn't want to take over the channel controlling memory refresh (for obvious reasons), while the output line of the other channel is connected directly to the PC speaker and can't be made to generate an interrupt.

Listing 1 contains the input queue manager timer control routines. When the machine starts up, PIT channel 0 is programmed by the PC `BIOS` to interrupt the machine 18.2 times a second (every 54.9 ms). When the input queue manager timer is initialized, it first reprograms this channel to a 1 ms resolution, which causes interrupts to occur 1,000 times a second. It then replaces the timer interrupt handler with our own specially constructed interrupt handler. Our handler counts the number of clock ticks that have elapsed and calls the `BIOS` timer interrupt handler every 54.9 ms (approximately). In addition, our handler counts down each of the 16 alarms and, if appropriate, generates timer events for alarms with expired counts. Alarms in continuous mode are reinitialized after their timer events have been enqueued.

Mouse Events

Until now, whenever we've wanted to gather events from a PC input device, the input queue manager has had to usurp control of that device at the interrupt handler level. While this approach certainly works, it is not a method I recommend to novices. Even veteran PC programmers can run up against unanticipated problems leading to frustrating hours of debugging.

Luckily for us, we don't need to use this method to take control of the mouse. When mouse support was added to PCs, someone took the time to design a well-thought-out and complete API to it. The mouse API is available after the mouse device driver has been loaded (this driver is usually supplied with the mouse hardware). Once loaded, the mouse driver API is available via interrupt `33h` and provides an interface similar to MS-DOS's interrupt `21h` services.

The mouse API provides a number of features we'll make use of: mouse detection and initialization, movement range and sensitivity adjustment, and, most importantly, the ability to specify a mouse event handler.

Mouse Initialization

Our initialization routine will configure the mouse to take full advantage of the desired graphics mode and install our mouse event handler. Subsequently, any mouse events that we've requested notification on (mouse movement, button presses, and so on) will cause our mouse event handler to be called. From there, we can figure out the type of the mouse event and whether or not an `EVENT` object should be created for it and posted to the input queue.

Listing 2 contains excerpts from the input queue manager's mouse event code. `Initialize_mouse()` is called when the client specifies that mouse input is desired. Because the mouse handler supports any graphics mode for which a `draw_mouse()` routine has been implemented, the initialization routine accepts parameters specifying the current video mode.

Since the first pixel of all video modes is assumed to be at Cartesian coordinates (0,0), the initialization routine only requires that the user supply the maximum

Mike Michaels

Unfortunately, the keyboard is only half the battle. By controlling timer, mouse, and event queues, you can take absolute control of input.

Listing 1. Timer.C

```
#include "inpqpriv.h"
#pragma inline;

/*
** timer_events
*/

static void interrupt timer_events (void);

/* Local Data */

static void (interrupt *BIOS_timer_handler) (void) = NULL;
static u32      clock_ticks;
static u16      counter;

/* Interface Global Routines */

/*
** INPQ_set_alarm
*/

void INPQ_set_alarm (s16 alarmord, u16 milliseconds, BOOLEAN one_shot)
{
    if (alarmord < 0 || alarmord > LAST_ALARM)
        return;

    if (milliseconds <= 0)
    {
        alarminit[alarmord] = -1;
        alarm[alarmord] = -1;
    }
    else
    {
        if (!one_shot)
        alarminit[alarmord] = milliseconds;
        alarm[alarmord] = milliseconds;
    }
}

/* Library Global Routines */

/*
** initialize_timer
*/

BOOLEAN initialize_timer (void)
{
    if (!BIOS_timer_handler)
    {
        clock_ticks = 0;
        counter = PIT_FREQ / MILLISECOND;
        outp (PIT_COMMAND, PIT_SETFREQ);
        outp (PIT_CH0, counter & 0xFF);
        outp (PIT_CH0, (counter & 0xFF00) >> 8);
        BIOS_timer_handler = getvect (TIMER_INT);
        setvect (TIMER_INT, timer_events);
        return True;
    }
    return False;
}

/*
** release_timer
*/

void release_timer (void)
{
    if (BIOS_timer_handler)
    {
        outp (PIT_COMMAND, PIT_SETFREQ);
        outp (PIT_CH0, 0);
        outp (PIT_CH0, 0);
        setvect (TIMER_INT, BIOS_timer_handler);
        BIOS_timer_handler = NULL;
    }
}

/*
** timer_events
*/

static void interrupt timer_events (void)
{
    u16     i;
    EVENT *e;

    // adjust the count of the clock ticks
    clock_ticks = clock_ticks + counter;

    // time to call the BIOS timer to update the time of day?
    if (clock_ticks >= BIOS_COUNT)
    {
        // adjust clock tick count and call BIOS timer interrupt han-
dler
        // (don´t need to acknowledge interrupt since BIOS handler
will.)
        clock_ticks -= BIOS_COUNT;
        asm {
                pushf
                call    dword ptr BIOS_timer_handler
        }
    }
    else
    {
        // acknowledge interrupt
        outp (PIC_REGISTER, NONSPECIFIC_EOI);
    }

    // process count-down timers, post events if appropriate
    for (i=0; i<=LAST_ALARM; i++)
        if (alarm[i] > 0)
        {
            --alarm[i];
            if (!alarm[i])
            {
                // queue timeout event
                e = allocate_event ();
                e->type = timer_alarm;
                e->data.timer.alarm = i;
                alarm[i] = alarminit[i];
            }
        }
}
```

(x,y) coordinates of the mode. Pay close attention to this detail; the maximum (x,y) coordinate of a video mode is not the resolution of the mode. For example, Mode 13h has a resolution of 320 by 200, but its maximum coordinate is (319,199). We also need to know how many bytes make up a single scanline in the current mode because drawing the cursor is an inherently rectangular operation.

Finally, to support double buffering (Mode 13h) and page flipping (Mode X), we need a pointer to a routine that returns the current frame buffer address. This address is returned as a far pointer because it is not necessarily going to be pointing to video memory.

The first thing the initialization routine does is grab a pointer to the InDOS flag. This flag is nonzero when a DOS `int 21h` function is currently processing. If this flag is set when we enter our mouse event handler, we immediately exit back to the mouse driver (though the situation probably never occurs, it's best to check for it).

Next, we ensure that the video system is in graphics mode. The input queue manager doesn't support text mode mouse cursors. After this, we actually move on to initializing the mouse. Because the input queue manager overrides a number of functions provided by the mouse driver API, the names of all the functions that actually talk to the mouse device driver are prefixed with an `int33h_`. The `int33h_init_mouse()` routine requests the mouse driver API determine if a mouse is present and, if so, reset it to default operational values. If the mouse was properly initialized, this routine returns a nonzero value.

If the mouse was successfully initialized, the mouse cursor interface is initialized with a call to `MCITF_init()`. Here, all the video mode parameters required by `initialize_mouse()` are passed on to be stored for later use. We will discuss the mouse cursor interface and its associated routines later.

By default, the mouse device driver assumes a graphics resolution of 640 by 200. Because this is almost certainly not the desired resolution, we call `int33h_set_mouse_limits()` to adjust the limits and sensitivity of the mouse. This routine makes three calls into the mouse driver API to set the horizontal limits (`0..max_x`), the vertical limits (`0..max_y`), and the sensitivity of the mouse.

The hardware mouse driver also initializes the mouse position at the center of the screen. This position is no longer going to be valid. Therefore, the initialization routine calculates a new center location from the maximum horizontal and vertical positions and uses `int33h_set_mouse_position()` to move the cursor there.

Table 1. Register State when Mouse Event Handler is Called

ax	Mouse Event Flags	Result
	Bit 0	Mouse movement
	Bit 1	Left button down
	Bit 2	Left button up
	Bit 3	Right button down
	Bit 4	Right button up
	Bit 5	Center button down
	Bit 6	Center button up
	Bits 7 to 15	Reserved (0)
bx	Button State:	Result
	Bit 0	Left button is down
	Bit 1	Right button is down
	Bit 2	Center button is down
	Bits 3 to 15	Reserved (0)
cx	Horizontal (x) pointer coordinate	
dx	Vertical (y) pointer coordinate	
si	Last raw vertical mickey count	
di	Last raw horizontal mickey count	
ds	Mouse driver data segment	

The default configuration of the mouse driver API does not provide a mouse event handler. The client is expected to poll for changes in the mouse's state. Because we want to be notified automatically when the mouse state changes, we are going to introduce a mouse event handler into the equation.

`Int33h_set_event_handler()` registers a mouse event handler with the mouse device driver. When a mouse event handler is registered with the mouse device driver, a mask of the events that the handler is interested in is supplied. Subsequently, the mouse device driver will only call the mouse event handler when those specific events occur. The bit mask the input queue manager passes to the mouse device driver indicates that it will be handling all mouse events.

Finally, we register the default cursor shape with the mouse cursor interface via the `MCITF_shape()` routine. The last thing the initialization does before returning a `True` indication is to set the `mouse_visible` variable to -1 (for reasons I'll explain).

Mouse Visibility

The mouse cursor is only made visible when the `mouse_visible` variable transitions from -1 to 0 on a call to `INPQ_show_mouse()`. Subsequent calls to `INPQ_show_mouse()` have no effect on this value. Calls to `INPQ_hide_mouse()`, on the other hand, always decrement this variable. If the value transitions from 0 to -1, the visible mouse cursor is hidden. Thus, every call to `INPQ_hide_mouse()` must be matched with a call to `INPQ_show_mouse()` before the mouse will again become visible, while excess calls to `INPQ_show_mouse()` are ignored. These semantics match those defined by the mouse driver API.

Calls to `INPQ_show_mouse()` and `INPQ_hide_mouse()` don't just affect the visibility of the mouse cursor, they also affect whether mouse events are gathered. The mouse event handler checks the `mouse_visible` variable before the mouse cursor update and if it's nonzero, exits the mouse event handler altogether.

In some situations, this is the desired effect, but if we're double buffering or page flipping, we want mouse events enabled even when we've hidden the mouse cursor so that we can draw the next frame. To support this, the input queue manager API includes two functions to show and hide the mouse cursor without affecting event gathering: `INPQ_obscure_mouse()` and `INPQ_unobscure_mouse()`.

The obscure and unobscure functionality is implemented at the lowest level,

Listing 2. Excerpts from Mouse.C (Continued on p. 45)

```
#include "inpqpriv.h"
#include "mcitf.h"

/*
** initialize_mouse
*/

BOOLEAN initialize_mouse (s16 max_x, s16 max_y,
        u16 bytes_per_scanline,
        u32 (far *active_page) (void))
{
    union REGS  regs;
    struct SREGS    sregs;

    segread (&sregs);
    regs.h.ah = GET_INDOS_FLAG;
    intdosx (&regs, &regs, &sregs);

    InDOS = MK_FP (sregs.es, regs.x.bx);

    // make sure that we are in graphics mode (this handler
    // is for use with the GFX library and doesn´t support
    // text cursor updates).

    regs.h.ah = GET_VIDEO_MODE;
    int86 (VIDEO_INT, &regs, &regs);

    if (regs.h.al <= 4 || regs.h.al == 7) // modes 0-4 & 7 are text
modes
        return False;

    if (!int33h_init_mouse ())
        return False;

    // call mode specific mouse initialization
    MCITF_init (max_x, max_y, bytes_per_scanline, active_page);

    int33h_set_mouse_limits (max_x, max_y, 8, 8);

    current_x = (max_x + 1) / 2;
    current_y = (max_y + 1) / 2;
    int33h_set_mouse_position (current_x, current_y);
    int33h_set_event_handler (ALL_MOUSE_EVENTS, mouse_events);
    MCITF_shape (
        default_cursor, default_width, default_height,
        default_hotspot_x, default_hotspot_y, 1, 0);

    mouse_visible = -1;
    return True;
}

/*
** INPQ_show_mouse
*/

void INPQ_show_mouse (void)
{
    // Only show the mouse cursor if the mouse_visible variable
    // transitions from negative to zero.
    if (mouse_visible == 0 || ++mouse_visible != 0)
        return;
    MCITF_draw (current_x, current_y);
}

/*
** INPQ_hide_mouse
*/

void INPQ_hide_mouse (void)
{
    // only hide the mouse cursor if the mouse_visible variable
    // transitions from 0 to -1.
    //
    if (--mouse_visible == -1)
        MCITF_draw (-1, -1);
}

/*
** INPQ_mouse_visible
*/

BOOLEAN INPQ_mouse_visible (void)
{
    return mouse_visible == 0;
}

/*
** INPQ_obscure_mouse
*/

void INPQ_obscure_mouse (void)
{
    extern BOOLEAN MCITF_obscure;

    if (!INPQ_mouse_visible ())
        return;

    if (!MCITF_obscure)
    {
        MCITF_draw (-1, -1);
        MCITF_obscure = True;
    }
}

/*
** INPQ_unobscure_mouse
*/

void INPQ_unobscure_mouse (void)
{
    extern BOOLEAN MCITF_obscure;

    if (!INPQ_mouse_visible ())
        return;

    if (MCITF_obscure)
    {
        MCITF_obscure = False;
        MCITF_draw (current_x, current_y);
    }
}

/*
** mouse_events
*/

static void far mouse_events (void)
{
    // on entrance, registers are set up as follows:
    //
    //  ax = mouse event flags:
    //  0 = mouse movement
    //  1 = left button down
    //  2 = left button up
    //  3 = right button down
    //  4 = right button up
    //  5 = center button down
    //  6 = center button up
```

Listing 2. Excerpt from Mouse.C (Continued on p. 46)

```
    //  7-15 reserved (0)
    //  bx = button state
    //  0 = left button is down
    //  1 = right button is down
    //  2 = center button is down
    //  3-15 reserved (0)
    //  cx = horizontal (X) pointer coordinate
    //  dx = vertical (Y) pointer coordinate
    //  si = last raw vertical mickey count
    //  di = last raw horizontal mickey count
    //  ds = mouse driver data segment

    s16 x, y;
    s16 region;
    u16 event_flags, event_mask, button_state;

    asm {
    push    ax
    push    bx
    push    cx
    push    dx
    push    si
    push    di
            push    ds
            push    es

            mov event_flags, ax
            mov button_state, bx
            mov x, cx
            mov y, dx

            mov ax, DGROUP
            mov ds, ax
    }

    if (InMouseEvent) goto clear_mouse_event;
    InMouseEvent = True;

    asm {
            mov cx, x
            mov dx, y
            mov current_x, cx
            mov current_y, dx

            // see if the cursor is visible to the user (mouse_visible ==
0)
            test    mouse_visible, 0xFFFF
            jz      draw_cursor
            jmp clear_mouse_event
    }

    draw_cursor:
    MCITF_draw (x, y);

    if (*InDOS) asm jmp clear_mouse_event

    asm {
            // register event for this interrupt (if applicable)
            push        y
            push        x
            call        far ptr in_region
    add     sp, 4
            cmp     ax, -1
    jne     check_events
    jmp     clear_mouse_event
    }

    check_events:
    asm {
```

```
            mov region, ax

            mov ax, 0
            mov dx, event_flags
            test    dx, EF_MOUSE_MOVEMENT
            jz      button_down
            or      ax, mouse_move
    }

    button_down:
    asm {
            test    dx, EF_MOUSE_BUTTON_DOWN
            jz      button_up
            or      ax, mouse_down
    }

    button_up:
    asm {
            test    dx, EF_MOUSE_BUTTON_UP
            jz      verify_event
            or      ax, mouse_up
    }

    verify_event:
    asm {
            test    ax, events_enabled
            jnz queue_event
            jmp clear_mouse_event
    }

    queue_event:
    asm {
            mov event_mask, ax

            call    far ptr allocate_event // returns far ptr to
event in dx:ax
            cmp     dx, 0                  // null pointer returned?
            je      queue_overflow         // drop event

            mov bx, ax                     // move dx:ax => es:bx
            mov es, dx

            mov ax, event_mask
            mov es:[bx].type, ax

            mov dx, event_flags
            mov cx, button_state

            mov ah, 0
            test    dx, EF_LEFT_BUTTON
            jnz set_left
            test    cx, BS_LEFT_DOWN
            jz      right_button
    }

    set_left:
    asm {
            or      ah, LEFT_BUTTON
    }

    right_button:
    asm {
            test    dx, EF_RIGHT_BUTTON
            jnz set_right
            test    cx, BS_RIGHT_DOWN
            jz      center_button
    }

    set_right:
```

within the routine that draws the mouse cursor to the frame buffer. The show and hide functionality, on the other hand, is implemented at the topmost level, within the input queue manager itself.

The obscure and unobscure model does not follow the mouse driver API nesting rules. Obscurity is maintained as a Boolean value by the mouse cursor interface rather than as a counter.

The Mouse Event Handler

Now, let's examine the actual mouse event handler. This handler is called by the mouse driver any time a mouse event occurs. Table 1 defines the state of the CPU registers when this routine is called. Because the interface is register based, I decided to code the mouse driver in inline assembly language.

The first thing our mouse event handler does is save the input data to the stack. The x86 processors have never had enough registers, so it just doesn't pay to keep values in them over the course of a routine as large as this one.

One thing to pay close attention to here is that the mouse device driver calls our mouse event handler with its own data segment in `DS`. To access our own program specific data, we must reset the data segment register. This is easily accomplished by moving `DGROUP`, the linker symbol for our data segment, into `DS`.

Next, we do a little cursor housekeeping. The current x and y coordinates of the mouse cursor are copied to a pair of global variables. These variables maintain the input queue manager's concept of the current mouse cursor location. We also draw the mouse cursor at this point if it is currently visible (or erase it if it's still visible when it shouldn't be).

Now we're ready to determine if we need to translate the mouse event into an input queue event. The first check we make is to determine if we are within a region the user registered to receive mouse events from.

If the current mouse event has occurred within a defined mouse region, the region number is saved away for possible later use. Next, we build the event type word and check to see if the client requested notification for these specific events. If so, an event will be posted to the input queue. Figure 2 depicts the structure of a mouse event.

The event is constructed and enqueued by calling `allocate_event()` (I described this routine in my previous article). The event structure returned by this routine is then filled in with the event type, current state of all the buttons, the region within which the event occurred, and the current (x,y) coordinates of the mouse cursor.

Listing 2. Continued from p. 45

```
    asm {
            or      ah, RIGHT_BUTTON
    }

    center_button:
    asm {
            test    dx, EF_CENTER_BUTTON
            jnz set_center
            test    cx, BS_CENTER_DOWN
            jz      set_attributes
    }

    set_center:
    asm {
            or      ah, CENTER_BUTTON
    }

    set_attributes:
    asm {
            mov al, byte ptr region
            mov es:[bx].attr, ax

            mov ax, x
            mov es:[bx].data, ax

            mov ax, y
            mov es:[bx].data+2, ax

            jmp clear_mouse_event
    }

    queue_overflow:
    asm {
            inc lost_input
    }

    clear_mouse_event:
    InMouseEvent = False;

    asm {
            pop es
            pop ds
            pop di
            pop si
            pop dx
            pop cx
            pop bx
            pop ax
    }

    InMouseEvent = False;
}
```

Mouse Regions

I'd like to spend just a little time talking about mouse regions and their intended use. The first thing I'd like to point out is that mouse regions, unfortunately, are brain-dead.

Mouse regions are rectangular areas of screen space. When the mouse cursor is within one of these regions, all mouse events that have been requested by the client are returned as input queue events. Likewise, if mouse events occur outside any mouse region, regardless of whether the client enabled the mouse event, no input queue events are posted.

On the plus side, mouse regions are great for reducing the number of mouse events enqueued. Most of the time, there is no reason to handle movement events, it just wastes processor cycles. Mouse regions, by their very nature, eliminate the majority of excess movement events. Mouse regions work great for dialog boxes and static introduction screens where there are a number of boxes or buttons that can be poked and prodded with the mouse cursor.

Where mouse regions fail, though, is with any sort of animation. This failure is due mainly to the fact that mouse regions cannot overlap. The current implementation has no concept of overlapping regions and won't allow a region to be defined if it overlaps an existing region. If you want to animate a sprite over some region of the screen, do not expect to be able to define an object that the sprite can interact with anywhere within the same region. It's pretty obvious that mouse regions aren't particularly useful.

With these caveats in mind, we'll move on to the actual implementation of mouse regions.

Mouse regions are maintained as a stack of region lists. When `INPQ_push_mouse_regions()` is called, a new entry is placed on top of the region stack. The new entry contains an empty region list. `INPQ_define_region()` is then called with the upper-left and lower-right screen coordinates of the region being defined. If the region doesn't overlap with a previously defined region, it is added to the current region list. When you're done with the current list of regions, INPQ_pop_regions() removes and discards the list of regions on top of the stack. By default, the input queue manager defines one region list containing a single region encompassing the entire visible screen. This region is never popped by INPQ_pop_regions().

The Mouse Cursor Interface

The mouse cursor interface defines a mode-independent API that allows a program to control how the mouse cursor is displayed. The mouse cursor interface comprises three functions: MCITF_init(), MCITF_shape(), and MCITF_draw(). I've provided implementations of these three functions for Mode 13h and Mode X.

All the mouse cursor interface routines are written in assembly language. There really is no good reason that the first two routines weren't written in C. They would certainly be more readable if they had been. But when I started the implementation of the input queue manager, quite a bit was written in assembly language, and I just haven't seen any pressing reasons to convert these routines.

Listing 3 contains the code to initialize the mouse cursor interface as well as the code to change the mouse cursor's shape and hotspot. MCITF_init() initializes the mouse cursor interface by storing away the current video mode information.

MCITF_shape() updates the bitmap (or masks) that control how the mouse cursor is displayed. This routine accepts mouse cursor shapes specified in one of two formats: one bit per pixel or eight bits per pixel.

In one-bit-per-pixel mode, the bitmap pointer points to the start of two concatenated masks. The first mask is ANDed with the screen contents to create a "hole" in the current image. The mouse cursor is then displayed within the hole by XORing in the second mask. To minimize the amount of calculation required when the mouse cursor is actually being displayed, the AND and XOR masks are expanded into eight-bits-per-pixel representations as they are copied to the mouse cursor save buffers within the mouse cursor interface. Currently, the mouse cursor interface doesn't know how to deal with 16- or 24-bit color.

In eight-bits-per-pixel mode, which I've dubbed direct mode in the source code, the bitmap pointer points to a block of memory that can be transferred verbatim to video memory. This block of data is copied without translation to the mouse cursor save buffer.

One-bit-per-pixel mode is the classic method of drawing a mouse cursor. If you use the change cursor shape routines available via the mouse driver API, this is the format that is expected. Eight-bits-per-pixel mode was added because it has become quite common in point-and-click style games to change the mouse cursor to some arbitrary bitmap that isn't representable using the AND/XOR method of drawing the cursor.

Drawing the Mouse Cursor

The MCITF_draw() routine is called to erase the previous mouse cursor from the current frame and (possibly) redraw it at another location. If you examine the code in Listing 3, you'll notice that while the listing is long, it's actually divided into three distinct parts:

- Common initialization code
- Mode-specific code
- Common exit code.

This file must be compiled separately for Mode 13h and Mode X. Ideally, you'll choose one mode or the other without needing to switch between them in the middle of your program. The makefiles, provided in the archive, build both types of libraries for you.

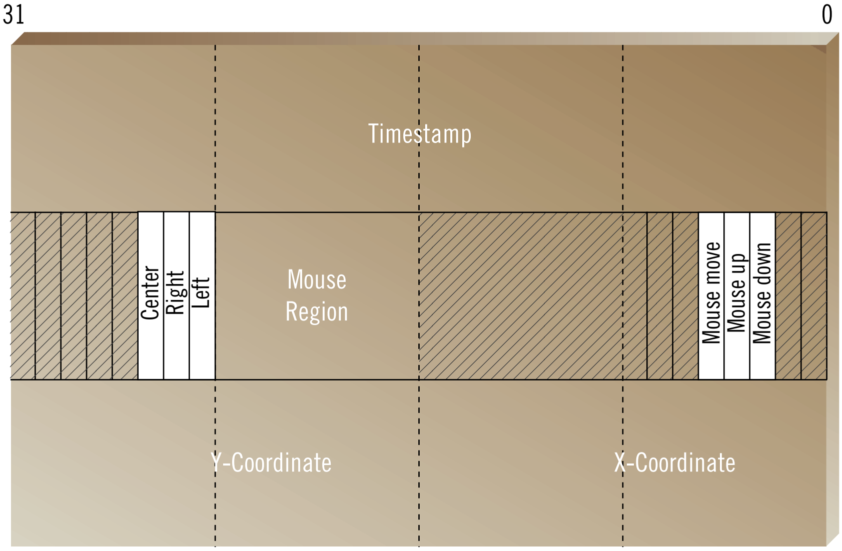
Let's start at the top. After saving away registers and making sure we know what our data segment is pointing to, we check to see if we're being called recursively or if the mouse cursor is currently obscured. Both conditions cause immediate exit.

Next, we determine where our frame buffer is by calling the user supplied active_page() routine (passed as a parameter to MCITF_init()). This routine returns a pointer to the current frame buffer. This can be in main memory or somewhere in video memory, it really doesn't matter to the MCITF_draw() routine. What is of concern to us is that, for every allocated page, we have a separate save space to store the previous background and cursor clipping information.

I designed the interface with the expectation that, in Mode 13h, double-buffering would be available through the use of a single main memory buffer. In Mode X, I assumed double-buffering or triple-buffering would be available via page-flipping. Because up to three frames could be active simultaneously, I statically allocated space for three mouse save areas. These areas are initialized to -1 at startup.

The next section of code runs through these save areas to see if it can

Figure 2. Mouse Event Structure



Listing 3. Excerpts from MCIINIT.ASM and MCISHAPE.ASM (Continued on p. 49)

```
;**
;** void MCITF_init (s16 max_x, s16 max_y,
;**             s16 bytes_per_scanline,
;**             u32 (far *active_page) (void))
;**

_MCITF_init proc far
        arg _max_x:word, _max_y:word, _bytes_per_scanline:word,\
            _active_page:dword

        push    bp
        mov bp, sp

        mov ax, _max_x
        mov max_x, ax
        mov ax, _max_y
        mov max_y, ax
        mov ax, _bytes_per_scanline
        mov bytes_per_scanline, ax
        mov eax, _active_page
        mov active_page, eax

        pop bp
        ret

_MCITF_init endp

;**
;** BOOLEAN MCITF_shape (u8 far *bitmap,
;**                 u16 width, u16 height,
;**                 u16 hotspot_x, u16 hotspot_y,
;**                 u16 bits_per_pixel, u8 transparent)
;**

_MCITF_shape proc far
        arg bitmap:dword, width:word, height:word, \
                hotspot_x:word, hotspot_y:word, bits_per_pixel:word,\
                transparent:word
        local   mask_size:word, direct:word=AUTO_SIZE

        push    bp
        mov bp, sp
        sub sp, AUTO_SIZE

        push    di
        push    si
        push    ax
        push    bx
        push    cx
        push    ds

        mov ax, width
        cmp ax, MAX_CURSOR_WIDTH
        jg      @@error_exit
        mov cursor_width, ax

        mov ax, height
        cmp ax, MAX_CURSOR_HEIGHT
        jg      @@error_exit
        mov cursor_height, ax

        mov ax, hotspot_x
        mov bx, hotspot_y
        mov hot_x, ax
        mov hot_y, bx

        mov ax, transparent
        mov mtransparent, al
```

```
        mov ax, seg and_mask
        mov es, ax
        mov di, offset and_mask

        mov ax, cursor_width        ; calculate number of bytes
        mov cx, cursor_height       ; in the packed representation
        mul cx
        mov mask_size, ax

        mov direct, 0

        lds si, bitmap

        cmp bits_per_pixel, 1
        jne @@eight_bits_per_pixel

@@one_bit_per_pixel:
        ;
        ; es:di                     -> and_mask
        ; es:di+CURSOR_SAVE_SIZE    -> xor_mask
        ; ds:si                     -> and bitmap
        ; ds:si+bx                  -> xor bitmap
        ;
        mov cx, ax
        shr cx, 3
        mov bx, cx      ; cx is count, bx is offset
        mov al, 80h
@@set_mask:
        test    byte ptr [si], al
        jnz @@set_and_mask
        mov byte ptr es:[di], 0
        jmp @@do_xor_mask
@@set_and_mask:
        mov byte ptr es:[di], 0FFh
@@do_xor_mask:
        test    byte ptr [si+bx], al
        jnz @@set_xor_mask
        mov byte ptr es:[di+CURSOR_SAVE_SIZE], 0
        jmp @@next_mask
@@set_xor_mask:
        mov byte ptr es:[di+CURSOR_SAVE_SIZE], 0Fh
@@next_mask:
        inc di
        shr al, 1
        jnz @@set_mask

        dec cx
        jz      @@fini

        mov al, 80h
        inc si
        jmp @@set_mask

@@eight_bits_per_pixel:
        ; In 8-bits/pixel mode, there is no and/xor masking going on.
We just copy the bytes as-is to the and_mask location and put them
        ; directly to the display when we draw the cursor.
        ;
        ; es:di -> and_mask
        ; ds:si -> bitmap
        ;
        cmp bits_per_pixel, 8
        jne @@error_exit

        mov cx, mask_size
        rep movsb

        mov direct, 1
        jmp @@fini
```

match the current frame address, referred to as the “current page,” with a previously stored page value. If it encounters a -1 page value before it encounters a matching page, it knows that the page has never been seen before and can be allocated to the current save slot.

Once we know where the save location for the current frame is, we mirror its contents to a series of stack variables. Why? Because we don’t have enough registers that can be used as base registers in 16-bit mode (the same problem we encountered when we were talking about the mouse event handler). Instead, we get to waste lots of time copying data to the stack and later copying it back to its save slot when a mouse is drawn or erased. Once we’ve mirrored the data, it’s time to move on to actually erasing the previous mouse cursor.

For simplicity, the following discussion is going to concentrate on drawing the mouse cursor in Mode 13h. The structure of the code in the Mode X block is the same, it just has to do more work to draw to the video buffer.

The first thing we check is whether or not there is an existing cursor that needs to be erased. If the screen offset (stored in the cursor save area) is -1, there isn’t a mouse cursor to erase. The screen offset will be -1 in only two cases: the first time `MCITF_draw()` is called and when the cursor wasn’t drawn the last time the current page was updated.

Listing 3. Continued. from p. 48

```
@@error_exit:
        mov ax, 0
        jmp @@final_exit

@@fini:
        mov ax, 1

@@final_exit:
        pop ds

        mov bx, direct
        mov direct_draw, bx

        pop cx
        pop bx
        pop ax
        pop si
        pop di

        mov sp, bp
        pop bp
        ret

_MCITF_shape endp
```

If there is a cursor on the current page, all the required information is read from the cursor save variables, and a loop is entered to copy the save data from the cursor save area to the current page, effectively erasing the cursor from the page.

The drawing phase starts by writing a -1 to the screen offset and checking to see if we need to draw the cursor or not. If the x-coordinate is less than zero, the cursor is hidden, and we return immediately.

If the cursor isn’t hidden, we need to reset a number of upkeep variables to their default values. `Mouse_x` and `mouse_y` are the (x,y) offsets into the mouse cursor bitmap denoting the starting byte of the mouse cursor. These values may be modified by the clipping process. `Mouse_skip` is the difference between the width of the mouse cursor and the number of bytes per scanline. If we’ve just drawn a row of the mouse cursor, we can add this value to the current x coordinate to get to the first byte of the next row. Finally, `mouse_width` and `mouse_height` are the clipped width and height of the mouse cursor, respectively.

Before we can actually draw the mouse cursor into the frame buffer, we must clip it against the frame buffer’s boundaries. All clipping takes place in relation to the mouse cursor’s hotspot, which is not necessarily going to be at the upper left corner of the cursor rectangle. We must clip in relation to the cursor’s hotspot because the coordinate of the hotspot is returned to the client when the current mouse position is requested.

Clipping against the hotspot isn’t as bad as it sounds because we are able to clip the x- and y-axes independently. To clip along the y-axis, we first subtract out our y-axis hotspot position. If the resulting y coordinate is greater than zero, then the mouse cursor lies completely within the display area vertically.

If the adjusted y-coordinate is negative, then we need to “shear off” the top of the mouse cursor. We negate the y-coordinate to find out how many pixels we need to lose. This value is then added to the mouse_y variable and subtracted from the `mouse_height`. If the `mouse_height` goes to zero, the mouse cursor is totally obscured and need not be drawn.

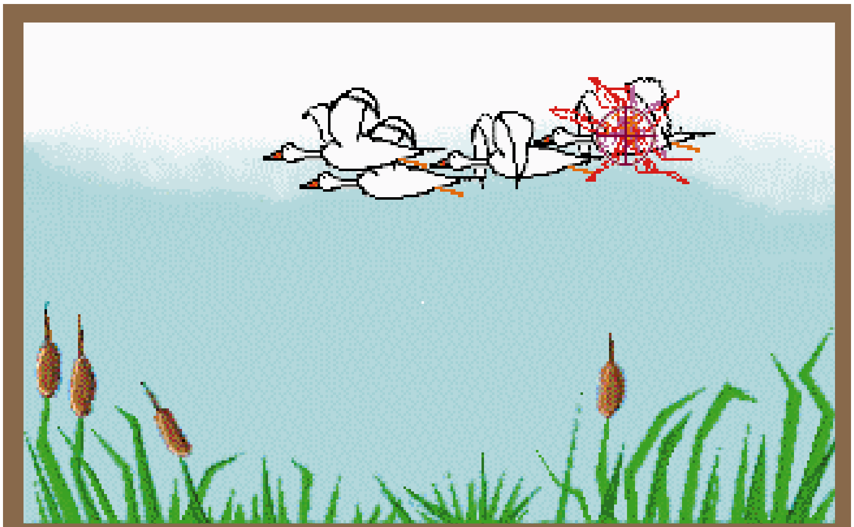
Next, we clip the y-coordinate against the bottom of the screen. We do this by adding the height of the mouse cursor to the previously calculated hotspot coordinate and check to see if the resulting coordinate exceeds our maximum y-coordinate. If so, we reduce the `mouse_height` by the required amount and again check to see if the height goes to zero, exiting if it does.

Otherwise, we move on to clipping along the x-axis, which I won’t go into because it is analogous to the discussion of y-axis clipping (replacing “top” references with “left” and “bottom” references with “right”).

After we’ve performed clipping, we need to figure out where in the frame buffer we need to draw the cursor. The `M13H_PIXEL_OFFSET` macro figures this out in an efficient manner. The offset is added to the current frame buffer address, and the result is placed in the screen offset save location. Next, the offset into the mouse cursor bitmap (or masks), specified by (`mouse_x`, `mouse_y`), is calculated using the `MOUSE_OFFSET` macro. With these two offsets calculated, the rest of the work is just setting up registers required for our draw loop. The last check made before we actually draw anything is whether or not we're in direct draw mode or not.

Both draw loops are nearly identical; for each pixel, the current page value is grabbed and saved in the cursor save area. New cursor data is placed in the display buffer, either directly or via the `AND/XOR` mechanism. While in direct draw mode, there is an additional transparent pixel check. If the current pixel value is equal to the specified transparent pixel value, nothing is drawn, and the background will show through the mouse cursor.

When the entire mouse cursor has been displayed, we jump down to the common exit code. This code saves all the stack values back to the appropriate cursor save location, pops all the saved registers from the stack, and returns.



This is a screenshot taken from the duck hunt program, which you can access on the *Game Developer* ftp site. It illustrates the principles discussed in this series.

This same code structure may be used to draw the mouse cursor in any mode. In addition to Mode 13h and Mode X, I've also implemented a mouse draw routine for 8-bit VESA modes. Unfortunately, the code is much too inefficient (and therefore embarrassing) to release.

User-Defined Events

If you examine the `EVENT_TYPE` structure, you'll notice that very few of the bits available are used for predefined events. In fact, the entire range from `0x0040` to `0x8000` is available for user-defined events.

A user-defined event may be anything you feel needs to be managed along with other input. For example, you might want to add user-defined events for joystick or HMD input.

To define an event, choose an ordinal from the unused range. I suggest starting with `0x8000` and working backward (if you're adding multiple events). This provides a vivid distinction between your events and predefined events when you're debugging your code. Next, you need to define structures analogous to the `_ATTR` and `_DATA` structures provided for keyboard, mouse, and timer events. That is, you need to define a 16-bit attribute structure and a 32-bit data structure that your event handler will fill in.

That's it for data definitions. Within your initialization code, you'll need to call `INPQ_enable_user()`, passing the `EVENT_TYPE` ordinal you've chosen and a pointer to an initialization routine. At a minimum, this routine needs to return a nonzero value to indicate that the initialization completed successfully and events of this type may be enabled. You may use this routine, of course, to load any interrupt handlers or do any other startup that your event requires.

You may then enqueue events by calling `INPQ_enqueue()`. You must have already constructed your data and attributes structures when you make this call. This routine will allocate an available event, fill in each of the fields and post the event to the input queue.

I've created a simple program that demonstrates user-defined events by capturing joystick state changes and translating them so that they result in mouse-like behavior. This example program may be found in the test subdirectory of the source archive, in the files JOYSTICK.H and JOYSTICK.C.

Duck Hunt

The duck hunt example program (HUNT.H and HUNT.C in the test subdirectory of the source archive) is a fairly complete example of everything that we've discussed in the last two articles (and some things we haven't). Basically, this program flies ducks across the screen and lets you target them with the cursor. Once targeted, you may blow the ducks away with reckless abandon.

You may use the mouse, the keyboard, or the joystick to target and shoot the ducks. Frame-rate limiting and frames-per-second counting are provided using two separate timer alarms.

All in all, this program is a fairly good demonstration of the capabilities and ease of use of the input queue manager, even if the "game" isn't particularly challenging.

Also, don't forget to take a look at GFX.C, a small graphics library for Mode 13h and Mode X that fully supports double-buffering and page-flipping (albeit, written in C and fairly slow). It's not much, but it can provide the basis for your own efforts along these lines.

Now You Can Manage

That's about all I have to say for now about the input queue manager. I plan on converting this code for use in a 32-bit flat model environment in my copious free time, but that's fodder for another article. I'd like to thank Mark Delmont for contributing the artwork for the Duck Hunt example. Without his help, we'd all be shooting white rectangles on a blue background (and I'd have to figure out a new name for the program)!

As always, the complete input queue manager source code may be obtained from the *Game Developer* ftp site (ftp://ftp.mfi.com in the gamedev/src/ directory) or on CompuServe (Game Developer Library of SDFORUM). If you don't have access to either resource, but you do have an e-mail address, I'd be happy to send you a uuencoded version of the archive. Just send mail to inpq-source@irvine.com with a subject line of "inpq source". ■

Mike Michaels is a senior software engineer at a small Irvine, Calif.-based company. While he searches in vain for a computer game company that can match his aerospace salary, he develops his own ideas at home in his nonexistent spare time. Contact him via e-mail at mike@irvine.com or through Game Developer *magazine.*

Back

Object Cache Management

Most video games are constrained by memory limitations. In a perfect world, you'd use millions of frames of animations, sound effects, tiles, textures, and the like, but often they just won't all fit in memory at the same time.

One solution is object caching. Object caching lets you have as much data as you need. If extra memory is available, object caching lets you take advantage of it. If extra memory is not available, however, it will still run (with lower performance). This is ideal in the PC world, where the amount of available memory available varies wildly from system to system. A cacheable object is defined as a "set of data" that can be recreated entirely, either by loading it from the disk or by recomputing it. This usually applies to write-once, read-many data found on CD-ROM and ROM cartridges.

The main component of a caching system is the cache manager, whose job is to provide virtualized accesses to objects, usually through an ID or handle. For example, to replace the conventional code:

```
image *im=load_image("filename");
im->put_image(screen,x,y);
```

a cache system would use:

```
cache_handle im=cache_manager.regis-
  ter_image("filename");
cache_manager.image(im)-
  >put_image(screen,x,y);
```

The function `cache_manager.image` does a quick check to see if the object is in memory and, if so, it returns a pointer to object data. If the object is not in memory, it reads it from the disk and returns a pointer to the newly loaded object. If there is not enough memory to load up the new object, the cache system scans all the cacheable objects and frees the oldest one. The cache manager keeps freeing the oldest cached object until memory can satisfy the allocation request. You can quickly access cache items using a lookup table to the ID.

Dealing with Time Overflow

Each time an object is accessed, the time marker is incremented and then stored in the `last_accessed` field of the cache item. If the `last_accessed` field is a `short` (2 bytes), you can only access something 65,536 times before this overflows; but if it is a 4-byte `long`, you can have 4 billion accesses. If an overflow occurs, the system will still work, but it will work poorly. Let's take a look at an overflow situation. First, there is an access of a large, infrequently used data item at `time = 65,535`. We increment `time` and get an overflow, setting `time` back to 0. Then we do two more accesses at `time = 0` and `time = 1`.

Now let's say at `time=1`, there is not enough memory to load the object. The cache system goes to free the oldest object that appears to be the object from `time=0`, but is really the large object from `time=65,535`. The large object will never get freed! The disk will thrash, and the user will not think you are very cool.

There are two ways we can deal with this. First, you can check the `time` after each increment and see if it overflowed. This can be coded in assembly language as one instruction. If the time has overflowed, adjust all the cache object `last_access` times by dividing by some number (say the number 2). This will preserve the relative times for each object and give you more `time` values to work with.

Or, you can check `time` periodically from within the game. When it goes above a threshold value, divide all the cached objects time value by 2. This solution eliminates the need to compare each object access, and it is preferable for extreme speed-intensive needs, but it also creates the possibility that `time` will overflow between checks.

Memory Management and Fragmentation

If you use a conventional memory manager you will have problems with memory constantly freeing and reallocation blocks of memory and thereby fragmenting. The memory space becomes fragmented with little holes from cache objects that have been freed. If your memory is tight, you will get to a point where the holes are no longer big enough to satisfy a request.

For example, suppose your memory layout looks like this:

```
static object
cached object
static object
cached object
static object
```

After freeing all the cached objects during a new allocation request, the layout looks like this:

```
static object
free
static object
free
static object
```

Although the two `free`s put together might satisfy the request, you cannot span your object across the static object in the middle (that is, at least not unless you depend on hardware paging).

The solution is easy. Keep two heaps. One heap is used to satisfy static allocation requests, and one for cached requests. You can simulate this with one heap by allocating the static requests at the bottom of the heap (growing up) and the cached requests at the top of the heap (growing down).

As the static grows upward and needs more space, the objects in the cached space can be freed without any problem.

Precache Modeling

When a game starts, how do you know which objects to precache (or load into the cache)? You can leave the cache empty, but the disk will be hit every time something new is accessed, which can be annoying to the user. Sometimes this is not noticeable. For example, Doom's programmers did not precache the sound effects, and it didn't negatively effect the game play.

If the game is simple enough, you can scan all the objects in a level and load up all the associated art and sound effects until it runs out of memory. This is fine in some games, but a more flexible game cannot predict at the start which objects will appear in certain levels.

A much better solution is to use play statistics. Have a level designer play a level several times through, and, as each

Jonathan Clark

Don't you hate that little green light and that little "ching-ching-ching" sound that means you are thrashing your hard drives? Your users certainly do.

cache item is accessed, increment its counter. At the end of the level, save all the cache IDs and their access counts. Now, the next time the level is played, you'll know which IDs to use. Sort the IDs in descending order and load the most frequently accessed IDs first. Keep loading until you're out of memory or until they're all loaded.

Cacheable Objects

It doesn't make sense to cache an object whose size is smaller than a cache entry.

A cache entry might look something like this:

```
struct cache_item
{
   void *data;
  // NULL if object not loaded
   long last_access;
  // time stamp of last access
   unsigned char type;
   short file_number;
  // index into filename list
   long offset;
  // offset into file to find this object
} ;
```

The size of this data structure is `4+4+1+2+4=15 +` figure in 4 to 12 byte overhead for the memory manager node information, so anything less than or equal to 27 bytes should not be cached.

When I was writing Abuse (a game produced by Crack dot. Com and the first affiliated game to be distributed by Origin), I used caching as much as possible. While Abuse was written primarily in C++, 8% of the code was written in LISP and interpreted during the game. A demo is available at http://crack.com or at ftp://ftp.odrom.com/pub/abuse (the demo includes a built-in level editor and LISP source code). Following is a list of cached objects I created while writing Abuse:

- Frames of animation
- Tiles and textures
- Images
- Particle animation data
- Sound effects
- Compiled LISP functions.

Abuse has a LISP interpreter, which compiles the program at run time into a tokenized form and writes this out to a cache file. Once this has been done, LISP functions can then be accessed just like regular cache objects.

Use caching for computable operations to speed up your game. When you cache the frame, duplicate it and make one frame reversed.

You can also use caching for computable operations to speed up your game. For example, say slamming an image right to left is faster than slamming it left to right, but you still need to draw the image both ways. When you cache in the frame, duplicate it and make one of the frames reversed.

Caching is also well applied to memory compression. In a system that can only access data through a CD-ROM, a disk hit can be very irritating to a user. How should you deal with this? You can store all your data in memory compressed and decompress it into cacheable objects as memory becomes available. Decompressing data is almost always faster than a CD-ROM hit, and if your data compresses well, this solution will work well for you.

Optimization

When dealing with inner loop material, you should save a pointer to a cache item instead of asking the cache manager to look it up everytime. For example, use:

```
image *img=get_img(tetxture);
for (int i=0;i<1000;i++)
   img->put_image(screen,x,y);
```

instead of:

```
for (int i=0;i<1000;i++)
   get_img(texture)->put_image
(screen,x,y);
```

Because you are not making any other cache request inside the loop, you can assume the object is still in memory. It is usually safe to assume the last several cache accesses will be in memory, but you can run into problems with sound or multithreaded programs. If you start playing a large sound in the background, and the cache manager frees and reallocates it before it is done playing, you will hear static. Either play small short sound effects or use a cache-locking system with sound callback.

The cache-locking system sets a bit in the cache item that says, "Don't free me." The sound system unsets this bit when the sound has finished playing. A lock and free system can be used to solve multithreading problems as well. Locks should not be obtained for long periods of time, though, because defragmantation cannot occur if an object is plugging a hole. Instead, use static allocation when you need "untouchable" objects for long periods of time.

You can also run into trouble if you work with large data objects. For example, suppose you want to blend two 640-by-480-by-256 images together and display it on the screen. The most efficient way would be :

```
char     *scan_line1=get_img(image1)
   ->scan_line(0),
  *scan_line2=get_img(image2)
   ->scan_line(0),
  *screen_scan_line=screen
   ->scan_line(0);
for (int count=w*h;count;count--,
  scan_line1++,
     scan_line2++,screen_scan_line++)
*screen=mix(*scan_line1,*scan_line2);
```

But if `image1` and `image2` don't fit into memory, you're in trouble. You will either have to break the images into smaller parts or write two mixing routines—an efficient one and another that runs in a low-memory environment by asking for the cache item on each iteration.

Finally, here are a few good tips I've picked up when writing cache systems and memory managers.

Optimize for small memory allocations. C++ programs tend to allocate and free small blocks of memory. I keep a list of stacks of pages for block sizes less than 128 bytes. Then, if an allocation is less than 128 bytes, the allocation time is usually a fast constant time. Memory profile your game and see what kind of allocations are going on and how often, then optimize for your needs.

Don't use virtual memory. Virtual memory involves writing pages to disk and reading them back. A cache system only needs to read, so it is much faster.

It's a good idea to word align all allocations to 4-byte boundaries. Though the x86 supports nonword aligned memory accesses, it does so only by reading twice. This can also make a significant difference if the stack is kept word-aligned.

A helpful debug tool is to zero out all freed memory. Then, whenever a reference to memory that is supposed to be free is used, a crash is likely to result and you can track it down easily.

Another useful debugging tool is to pass a string with each allocation which can be used to quickly point out the source of memory leaks or hogs. The `LINE` and `FILE` macros can even be used if you get lazy. These strings can be eliminated from the final executable by a define :

```
#define MY_malloc(size,string) my_malloc(size)
```

The two conditions above, bad references and memory leaks, are a big cause of bugs in games and are often hard to track down. These two methods can save you a lot of sanity and keep your game stable.

Don't forget today's freaks could be tomorrow's reality. 64-bit pointers seem overkill right now, but a breakthrough in memory production cost is all that is needed to move them from high-end systems to personal PCs. Think of the games you could write with over 4GB of memory. ■

Jonathan Clark is a partner of Crack dot Com and the author of Abuse. *Contact him via e-mail at jc@crack.com or through* Game Developer *magazine.*

Back

Gaming Warrior

Mike Michaels

Be the hero of your clan or just have fun blowing things up. Smart developers should take note of Mechwarrior 2's high-resolution graphics and movement effects.

Hello and welcome! This is my first time wielding the cleaver on the Chopping Block, so I thought I'd introduce myself. My name is Mike Michaels, and I'm a software engineer in Southern California. Part II of my series on PC input management appears in this issue on p. 40.

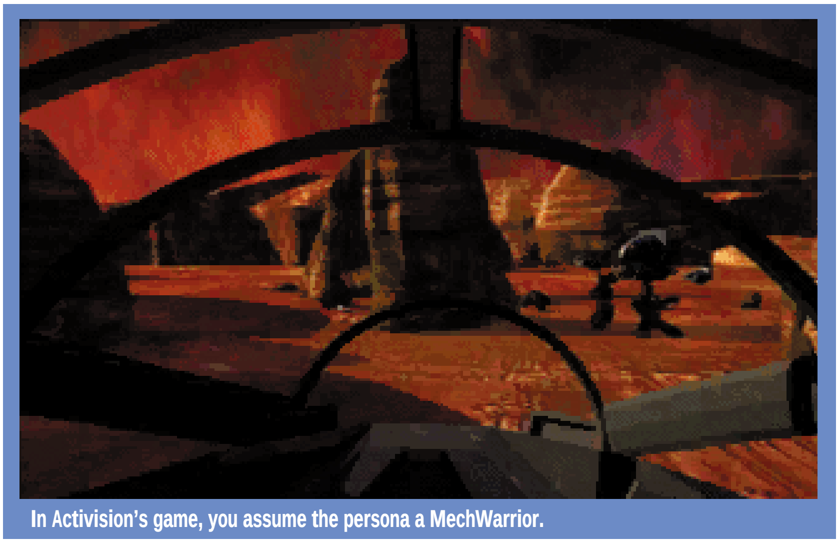
Now they've asked me to pen the Chopping Block. This was a difficult decision for me to make. On the one hand, writing a game review column has a certain attraction. On the other, I really wasn't interested in doing the column in the format that it had been rendered in previous issues. That is to say, I am not interested in hacking into a game's save file to figure out where and how everything is stored. If I'm struggling to finish my own games, why would I spend time hacking into other people's creations?

After a week of e-mail with the editor of this column, we agreed that a change in format might be appropriate. I'm going to try to structure these reviews at a higher level, while still maintaining a technical perspective.

Let's Get on with It!

This month we're going to take a look at Mechwarrior 2, a combat simulation game based on FASA's BattleTech Universe. In the game, you assume the persona of a MechWarrior, one of your clan's elite, who achieves honor, rank, and prestige by piloting the enormous BattleMechs (Mechs) into combat against the rival clan. If this is all Greek to you, don't worry. It really just gives you an excuse for running about in a mobile tin can and blasting everything in sight.

The game allows two modes of play. For the goal-oriented, there is a career path option that lets you undertake and complete missions that garner you prestige, honor, and rank amongst your peers. The ultimate reward for your selfless ser-



In Activision's game, you assume the persona a MechWarrior.

vice is to battle for and become the leader of your clan. If you couldn't care less about becoming a clan leader but you still want to blow away a few Mechs, the game provides an unrecorded combat mode where you lead up to two other Mechs into combat against a series of enemy Mechs.

The cut-scenes must be seen to be believed. Rendered in high resolution, they look like something you might see in a quality science-fiction movie. The only thing about them I didn't like was that I grew jealous of the freedom of movement that the choreographed Mechs had that I didn't—but I'm getting ahead of myself.

The actual simulation takes place in a real-time, three-dimensional, polygon rendered environment. Everything appears flat shaded, the lack of texture mapping probably accounting for the excellent frame rates—even at the highest resolution. Speaking of which, the game may be played in standard VGA (320 by 200) or either of two VESA high resolution modes (640 by 480 or 1024 by 768). I found the middle resolution to be the best visually: standard VGA resolution is just too chunky, while the highest resolution made the enemy Mechs look too small.

The engine is also capable of a few special effects. I noticed some light source shading as well as a fog-and-dust-type effect on some of the levels. Transitions from day to night seemed to be accomplished using progressive palette updates. There might be more effects in later levels, but my dexterity isn't such that I was able to find out.

There are a few anomalies with the rendering engine. In certain situations, it looked like enemy Mechs were walking on air. This usually occurred on a sloped surface and was probably a result of the lack of surface texture. Perhaps the most annoying artifact is the unrealistic perspective of objects at a distance. Rather than appearing gradually on the horizon, objects such as mountains, buildings, and enemy Mechs suddenly pop into existence as you near them. It's a little disconcerting to know you are being fired upon, but you can't see the enemy Mech at all until you've taken that last step.

You control your Mech using any number of standard input devices (and some nonstandard ones as well). You need the keyboard regardless of which optional input devices you choose. There are so many commands, they couldn't all be mapped to joystick or mouse buttons. The game supports input from a number of specialized joystick systems, including the Phoenix System, Microsoft Sidewinder 3D Pro, rudder pedals, and throttle control. In addition, Virtual I/O's i-glasses head-mounted display are supported.

The only real problem I had was the sensitivity of the controls. A fractional change in joystick position seemed to cause a disproportionate change in Mech position. I couldn't line up on an enemy Mech (even a stationary one) and hit it with a series of shots; I was never able to keep the target sighted in the reticle.

This problem was only aggravated by the higher-resolution modes. Couple the ultra-sensitivity of the controls with the longer lag times between handling user input (because it's taking longer to paint the screen), and you get a game unplayable in most people's resolution of choice. I would have appreciated some means of controlling joystick sensitivity.

If you know anything about the BattleMech universe, you're going to appreciate the effort designers went through to make each mission coherent and logical in the general framework. Those unfamiliar with the universe will be grateful the game was designed so you don't have to read the extremely long and sometimes cryptic details of what is going on in the clan war. In addition to detailed mission briefings, there is extensive background information to let you figure out where you and your clan fit in the grand scheme of things.

The game's designers have also done an excellent job capturing the physics behind Mech movement and combat. The cockpit bobs as your Mech walks or runs (à la Doom's floating gun affect). When you're running at full speed, it's much harder to make a sharp turn than when you're going half speed. When an enemy Mech fires and hits you, your cockpit rocks violently with the shock of the blast. Firing some of your weapons too often can cause your Mech to overheat and shut itself down—which can be fairly annoying if you have two or three enemy Mechs using you for target practice at the time.

MECHWARRIOR 2

Suggested Retail Price: $59.95
System Requirements: 486DX2/66, 8MB RAM, 45MB free hard-drive space, VGA or SVGA graphics, Soundblaster-compatible soundcard, joystick, double-speed CD-ROM.

Activision Los Angeles
10601 Wilshire Blvd. Ste. 1,000
Los Angeles, Calif. 90025
Tel: (310) 473-9200
Fax: (310) 479-4005
Web: http://www.activision.com/

Wrapping Up

Mechwarrior 2 is an enjoyable and (probably for some) addicting game. If you get past the sensitive controls, it's fun blowing other Mechs all over the landscape. The sound, cut-scenes, and transition scene graphics are excellent. While the polygon rendering engine is by no means a technological breakthrough, it's smooth, fast, and sufficient to let you immerse yourself in the BattleTech universe.

That's about it. Let me know what you think about the new direction this column is taking. Is it good, bad, would you rather we got rid of the it in favor of longer articles on perspective-correct texture mapping? There's been discussion of diversifying the column to include book reviews as well as reviews of game development tools and environments. I'd like to make this column as useful and entertaining as possible, so let me know what you would like to see. ■

Mike Michaels works for a small compiler company in Irvine, Calif. Though offered numerous jobs in the computer gaming industry, he has been unwilling or unable to take the pay cut involved in such a transition. He resides on the fringes of game development, living the life vicariously through friends on "the inside." Contact him via e-mail at mike@irvine.com or through Game Developer *magazine.*

A Question of Character

David Sieks

Computers make good (and bad) animation easier—that's why more people than ever are in the field. With competition this stiff, you need to have the best possible graphics to play the game well.

Someone, I wish I could remember who, once said that music lives in between all the notes. The comment has since, quite appropriately, been applied to the art of animation, as in "Animation lives in between all the frames." Beyond the Zen loopiness of the statement is an unavoidable kernel of truth; unavoidable, that is, for anyone who has attempted character animation. Although software tools can make it almost embarrassingly easy to, say, move a starship across the screen, depicting a character with convincing mass, momentum, and personality still calls for skills and techniques that animators have been refining throughout this century—since long before the advent of the computer.

It seems that many novice animators or nonartists involved in game production assume that technology has somehow removed all or most of the hurdles between them and snazzy animated routines. The truth is that it has made a lot of things easier, and one of those things is *bad animation*. Thanks to the computer, it's never been easier to create stiff, lifeless, uninspired animation; all too often, due to time constraints and laziness, that's what people are creating.

Doug Aberle, Master Animator at Will Vinton Studios—and perhaps best known to SIGGRAPH attendees as creator of the animated short *Fluffy*—observes that "the computer should be a tool, not a style. The basics of Squash and Stretch remain the same." Computoons president Bob Terrell (another Vinton alum) goes a step farther: "There should be a sticker on the software box that says Talent Not Included." If you're not already familiar with the principles of

"Martin [Hash] kids me that I used his spline-based modeling tool to create a polygonal character," Doug Aberle says of *Fluffy*, hit of the SIGGRAPH95 Electronic Theatre.

Sapphire Inc. (formerly Cygnus Multimedia) has moved from creating game graphics to producing entire games, though artists still outnumber programmers 3 or 4 to 1.

Squash and Stretch, or if you've been thinking the right software package would make your first animation project a snap, I hope you'll keep reading. I think I've got just about enough space here to impart a healthy respect for the work involved in character animation and to give beginners some idea of how to go about it.

Les Pardew, president of game production house Sapphire Inc. (where artists outnumber programmers nearly four to one), makes it quite clear why you should care: "Production values have gone up tremendously. To compete in this industry you've got to have really superior graphics or people just don't accept it as a game anymore." We've all seen too many games fat with fancy graphics but starving for gameplay, but we've also seen the extent to which high quality graphics can enhance the overall experience. All other things being equal, the better-looking game captures the player's attention and imagination, and solid character animation can make or break that overall look.

Know Your Roots

The key to good animation is in the very definition of the word. Though many would assume that "animate" means "to make move," its Latin root really means "to bring to life." The computer is good at making things move on the screen but making them seem alive is up to you. This can only come from an in-depth understanding of both traditional animation techniques and real-life movement.

As almost any animator will tell you, one good starting point is *The Illusion of Life* (Hyperion, 1981) by Frank Thomas and Ollie Johnston. Recently reissued, this massive tome contains the combined wisdom of these two renowned animators, who worked at Disney Studios from the mid 1930s until 1978, during which time they helped raise animation from a crude novelty to an art form. You may not be going after a Disney look, but this book covers all the basics every character animator should employ.

Another way to start gaining an appreciation for character animation is to really *watch* some of the classics: Disney films, of course, and the old Looney Toons and Tex Avery collections are all worth a close look, though you can skip the Hanna Barbera stuff. Classic live action films can help too: Aberle draws inspiration from the films of Laurel and Hardy, other animators often cite Buster Keaton and Charlie Chaplin, but what you choose to pay particular attention to really depends on what you plan to animate. I find Douglas Fairbanks and Errol Flynn great to watch for action. In general, older films tend to have longer takes, which let you watch the movement through its course, while modern films are composed of quicker cuts. Use a slow-framing feature or the frame advance on your VCR or laserdisk to observe every stage of the action, and try to isolate the extremes of motion. These correspond to the key poses from which an animator builds a routine.

Casting Call

Of course, character animation must start with a character, and that character starts with a design. The animator who has the luxury of designing his or her own characters deserves the envy of every Hollywood casting director, for therein lies the opportunity to literally create the perfect "actor" for the role. Less enviable is the animator who must bring to life a character designed by a nonartist. The difficulty in this not uncommon situation comes in matching the demands of the concept with the demands of the medium.

The first consideration in character design should be the game style and target audience. As Terrell points out, "Terms like 'cute' or 'scary' can mean vastly different things to different audiences." A more adult audience may be scandalized by the violent death scenes in a fighting game kids think is cool. What's deliciously creepy to a mature game player might cause nightmares in young children, while what kids find enjoyably spooky might seem simply corny to an older sibling. Keep in mind that much of the game's personality will reside in the depiction of its characters.

The player's view of the scene and of the characters in it is another important design consideration. How much of the character is seen—and from what angles—can help the animator determine what level of detail needs to go into various aspects of the design. As I'll discuss later, a character designed for side-scrolling gameplay can have very different needs than one featured in a highly detailed cinematic sequence. The key point is that the character must work from every angle that will be seen. Conversely, the animator can save precious time by not putting design effort into angles that won't be seen.

Before even turning to their sketchbook though, many animators will first sketch out ideas for the character's personal history. This isn't strictly the artist's job and may be unnecessary if a complete script has already taken care of these details or if the character is a straightforward monster or a hero who'll never be seen out of battle armor. Then again, such insight into the character can be helpful even in the latter cases: think of how much more interesting Tolkein's Gollum was because of his history.

Creating animation for the Cyclone Studios/3DO title Captain Quazar, Terrell found himself stopping production to flesh out the musclebound hero's background. "We did a full history for Quazar, including a lot of stuff that doesn't even really enter into the game. It was unorthodox to stop in the middle of the project to do it, and it did take up valuable time, but we felt it was necessary to get a good feeling for the character we were animating."

Next stop on the road to a good character design is the model sheet. By this time, the animator should have a very clear idea of the character's personality and of what sort of action will be called for. It's time to determine what the character looks like in detail. One common misstep is drawing the character standing rigidly at attention from square front and side views—unless that's appropriate behavior for the character. Rather, the model sheet should depict the range of attitudes, expressions, and general positions that will be called for in the animation.

Moving on from the model sheet is what is known as inspirational reference or atmosphere sketches. These are more fully realized drawings, paintings, renderings, or even sculptures depicting the character, often in a setting. Terrell, whose background is in claymation (at Will Vinton Studios he worked on the California Raisins and the Domino Pizza Noid), now does all his animation on the computer, but he still creates characters in clay for inspirational reference. "It helps for the artist to be surrounded with actual, physical things relating to the character," he notes. And though artists working for Animation Magic also do their animation on the computer, traditional hand-painted animation cels of the same characters hang on the office walls as inspiration.

Plan to be Spontaneous

Does it seem like I'm taking a long time to get around to talking about the act of animating? In a recent 3dSite IRC panel discussion on New Directions in Character Animation, Tim Johnson, Animation Director for Pacific Data Images, said, "I believe passionately that diving in leads to crummy animation. We all like to imagine ourselves as the great jazz improvisors of animation. Not true. If you don't act it out, draw it, think it through, your movement will probably suffer." The medium presents a paradox for an animator to overcome. An animated scene or routine is usually short and the action quick and full of life. But the process of making an animation is drawn-out and painstaking. Perhaps the greatest challenge is not to be defeated by that process, to resist being bogged down by preparations that can sap enthusiasm, and to reject the impulse to simply skip all the careful preparation and rush into animating the scene.

Though it is much more practical to revise a computer animation than is the case with either cel animation or claymation, it's still important to approach the scene with a well-thought-out plan in mind. While spontaneous creative flourishes will, hopefully, quite often enliven the actual animation process, the principle place and time for improvisation is in the mind of the animator and in the sketchbook, before ever bringing the project to the computer.

To hear Thomas and Johnston tell it, the halls and offices of Disney Studios were filled with animators acting out routines for their characters to get a feel for the movement, a tradition started there by Walt Disney himself. Aberle slyly refers to a reference tape of himself acting out the part of *Fluffy* (no, he won't show it to you). Even when the character is nonhumanoid it can help greatly for the animator to mimic the movement—as closely as possible, anyway—to get a better sense of the interplay of muscles and the shifting of balance. Throughout, special attention should be paid to anticipation and follow-through—the movements that lead up to and follow major actions. These are the small touches that make a movement more convincing and actually help to focus the audience's attention on the major action.

Once the movement is mapped out in the animator's mind, it's time to begin working out the key poses in a series of small, rough "thumbnail" sketches. When making such sketches, the artist should not start with the character's head, a common practice. Almost all movement originates in the pelvis or shoulders, and a sketch meant to convey motion should follow the focus of that action. At this stage, the drawing should depict mass and momentum more than surface detail. Disney encouraged animators to work in a quick, rough style, knowing the image retained more vitality this way. (Most of the "clean-up" was done later by junior animators.) Though a computer animator's sketches might contribute less directly to the final rendered product, capturing that energy beforehand is still a critical step.

Once the animator has expanded on the thumbnails to refine the key poses, the next step is to create a storyboard; a sort of visual map for the routine, progressing from pose to pose and taking framing into account. An important principle to keep in mind at this stage is *silhouetting*. That is, staging the action in such a way that it can be clearly seen by the audience, being careful not to allow important details or gestures to be muddied or hidden.

Squash and Stretch is one of the most basic principles of animation yet is unfortunately neglected by some computer animators. The idea is that living organisms—a group that includes most of our characters—are not as rigid in their movements as is a folding chair or a slide rule or any other inanimate articulated object. In the course of a movement muscles flex and extend, flesh sags or tightens, and these effects impart realism to an animation and can be exaggerated by the animator to enhance movement. However, it's relatively easy to model a character on the computer with limbs of the right proportions and joints in the right places, to move it around like a marionette and then wonder why the action looks stiff and unconvincing. If an artist has managed to capture the vitality and weight of a movement in the sketchbook phase, it would be a shame to abandon it at this stage. Squash and Stretch has its place even in three-dimensional modeling.

Once the animator has brought all this preparation to the computer, there's at least one more impulse to resist—the very natural wish to finally see everything rendered in sparkling three-dimensional detail. It's usually more economical to first render a rough pose test to check move-

Computoons designed and animated Captain Quazar and others for the 3DO title produced by Cyclone Studios. Animation included a 90-second rap video. Image courtesy of The 3DO Co.

ment and timing: forget about texture mapping and dramatic lighting, don't use antialiasing, don't even think about raytracing. A quick render will show how well the action plays and point out areas where the timing is off or the movement doesn't quite read. One of the things to really look for at this stage is a sense of solidity and weight to the character. This is one of the most difficult things to simulate well and will likely take careful tweaking.

The Play's the Thing

This all may sound overly ambitious for what is, after all, *a game*. Yet even if you eschew (gesundheit) fancy splash sequences, you should be wary of underestimating the importance of good character animation for gameplay routines. Gameplay is after all where the player will be spending the most time; if the character movement is *out* of character, it's just not as captivating.

To remain visually interesting, characters—especially the main character—should have a variety of movement routines: walking, jumping, tripping, and so on. One shortcut to take advantage of is the character's symmetry; the routines can often be mirrored, so that when flipped a left turn routine, for example, can also serve for a right turn. Though this may mean that your hero's blaster suddenly shifts from his right hand to his left, Terrell cautions the animator not to obsess. "As an animator, the lack of continuity was difficult to accept, but the game format allows you to hide a multitude of sins; sprites are only an inch or so on the screen and fewer frames are rendered. The game designers said not to worry, no one would notice, and you don't, really." Which is not to say that the Computoons artists let Captain Quazar get sloppy. With the lower frame count in gameplay routines, they took advantage of the opportunity to retouch images frame by frame.

Actually, two distinct three-dimensional models were created of Quazar: one for the cinematic sequences and another for gameplay with exaggerated features so that details would read in the smaller scale. The trick was to create models that were different yet still read easily as the same character, and the key to that was having intentionally designed a character that was appealing and still simple enough to be flexible.

How Much is the Free Lunch?

Inverse Kinematics is one of the buzzwords that struggling novice animators cling to faithfully, sure that it promises a future of hassle-free character animation. While Inverse Kinematics certainly has its uses and its devotees, it should be understood that it comes with its own challenges and frustrations, too. What Inverse Kinematics does is allow the animator to establish hierarchical chains; the shin bone's connected to the knee bone, knee bone's connected to the thigh bone, and so on, and establish movement parameters for those chains. Then by "tugging" on the foot, for example, you can watch the leg extend, pivoting at the hip and bending at the knee in a natural manner.

Natural, that is, if you've made all the right connections and set appropriate parameters. Inverse Kinematics is generally a scripted process, and getting everything connected just right can be an elaborate and time-consuming endeavor. While it's possible to get the same movement in an animation without using Inverse Kinematics, kinematics can definitely save time further down the road once an animator has established a library of parameters for different routines.

Though it's still a more or less higher-end fantasy, the mention of motion capture can also tend to invoke visions of flawlessly realistic character movement without all the work. Yet while many accomplished animators are intrigued by the possiblities, others suggest that at this stage in the technology it may take more effort to fix the captured data than to animate a scene from scratch. "Motion capture gets you about 60% of the movement," Sapphire's Pardew estimates, while the rest needs to be tweaked or completely redone by a skilled animator to look right. Motion capture is probably most useful for sports games calling for realistic movement. But for many games, realistic movement actually falls short of the mark. "That's not the point of animation," Tim Johnson states. "Great animation caricatures reality and makes it artful. It caricatures motion to make a dramatic or narrative point." Heroic, exaggerated movement is more the stuff of video games.

Ken Cope, Senior Artist at Acclaim Coin-Operated Entertainment puts forth his aphorism #3C: "The closer reality is approximated, the more glaring any discrepancy becomes." Maybe the real art of animation is to be convincing without being too realistic. After all, if reality were all that, who'd bother playing our games? ■

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