GAME DEVELOPER MAGAZINE

FEBRUARY 2002

UBM Tech

GAME PLAN

LETTER FROM THE EDITOR

The Joys of Self-Regulation

What kind of kid were you? Did your mom have to nag you endlessly to clean your room, your constant refusals erupting into all-out wars? Or perhaps you were cheerful and obedient, always cleaning your room as you were told. Sometimes, if you did it without your mom having to ask, she might give you a cookie.

The game industry got its cookie in December when the Federal Trade Commission issued its latest report to Congress on "Marketing Violent Entertainment to Children," a follow-up to a report submitted in September 2000 which criticized marketing practices in the music, film, and videogame industries. The conclusion was that since the first six-month follow-up report released in April 2001, the film and videogame industries have continued to improve their rooms' tidiness with respect to diverting their marketing of violent-themed or otherwise objectionable entertainment away from minors. (The music industry, apparently, is still running away from Mom, screaming and slamming doors, and seemingly oblivious of the fact that Mom usually wins in the end.)

The self-regulatory efforts of the game industry's trade organizations, in particular the Interactive Digital Software Association and the Entertainment Software Rating Board along with its Advertising Review Council, are commendable for willfully assuming responsibility for industry marketing practices while rightfully continuing to defend our First Amendment rights. What they've so deftly realized is that when you address one critique purposefully and with measurable success, you fend off outside agencies who would regulate this industry without its best business and creative interests in mind. Successful self-regulation also helps to dilute criticisms lobbed at other aspects of our trade. But we're not out of the public-opinion woods yet.

The most important thing for our industry to do now is to continue to defend itself against errors of fact in public opinion, such as the persistent and pernicious misperception that videogames are predominantly made for and played by children. When Australia's Office of Film and Literature Classification devised its first videogame rating system in 1994, "...concerns were expressed about the interactive nature of computer games and the possible adverse effects on children, who were seen as the primary target audience for computer games." (OFLC Discussion Paper, "A Review of the Classification Guidelines for Films and Computer Games," 2001). Later guidelines established a 15-and-over rating for some games, but decreed — in a triumph of ambiguity — "the stronger computer games are banned." (OFLC "Guidelines for the Classification of Computer Games," 1999).

The same week the FTC released its updated report in the U.S., copies of clearly mature-themed games such as GRAND THEFT AUTO 3 and GHOST RECON were reportedly being whisked from store shelves in some Australian jurisdictions, at the height of the Christmas shopping season, in order to undergo reclassification. The head of Sony Computer Entertainment Australia can tell a reporter for the Sydney *Herald Sun* that half of Playstation 2 owners are over 30 years old, yet the country's regulatory body concluded just a few years earlier that there was no need for a mature rating for games as there was for film, because games were for children. Clearly there is a lot of confusion coming out of and real money evaporating into this non-self-regulatory system.

Lessons abound about the virtues of successful self-regulation and the perils of yielding control to outside interests. The U.S.-based industry has fortunately been rewarded with praise for its efforts so far, but now is not the time to rest on our gossamer-thin laurels, nor will it ever be.

Jennifer

GameDeveloper
600 Harrison Street, San Francisco, CA 94107 t: 415.947.6000 f: 415.947.6090

Publisher
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL

Editor-In-Chief
Jennifer Olsen *jolsen@cmp.com*
Managing Editor
Tor Berg *tberg@cmp.com*
Production Editor
Olga Zundel *ozundel@cmp.com*
Art Director
Audrey Welch *awelch@cmp.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Daniel Huebner *dan@gamasutra.com*
Jonathan Blow *jon@bolt-action.com*
Hayden Duvall *hayden@confounding-factor.com*
Advisory Board
Hal Barwood LucasArts
Ellen Guon Beeman Beemania
Andy Gavin Naughty Dog
Joby Otero Luxoflux
Dave Pottinger Ensemble Studios
George Sanger Big Fat Inc.
Harvey Smith Ion Storm
Paul Steed WildTangent

ADVERTISING SALES

Director of Sales & Marketing
Greg Kerwin e: *gkerwin@cmp.com* t: 415.947.6218
National Sales Manager
Jennifer Orvik e: *jorvik@cmp.com* t: 415.947.6217
Senior Account Manager, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 415.947.6224
Account Manager, Northern California & Southeast
Susan Kirby e: *skirby@cmp.com* t: 415.947.6226
Account Manager, Recruitment
Raelene Maiben e: *rmaiben@cmp.com* t: 415.947.6225
Account Manager, Western Region & Asia
Craig Perreault e: *cperreault@cmp.com* t: 415.947.6223
Sales Associate
Aaron Murawski e: *amurawski@cmp.com* t: 415.947.6227

ADVERTISING PRODUCTION

Vice President, Manufacturing Bill Amstutz
Advertising Production Coordinator Kevin Chanel
Reprints Stella Valdez t: 916.983.6971

GAMA NETWORK MARKETING

Senior MarCom Manager Jennifer McLean
Marketing Coordinator Scott Lyon
Audience Development Coordinator Jessica Shultz

CIRCULATION

Game Developer is BPA approved.

Group Circulation Director Catherine Flynn
Circulation Manager Ron Escobar
Circulation Assistant Ian Hay
Newsstand Analyst Pam Santoro

SUBSCRIPTION SERVICES

For information, order questions, and address changes
t: 800.250.2429 or 847.647.5928 f: 847.647.5972
e: *gamedeveloper@halldata.com*

INTERNATIONAL LICENSING INFORMATION

Mario Salinas
t: 650.513.4234 f: 650.513.4482 e: *msalinas@cmp.com*

CMP MEDIA MANAGEMENT

President & CEO Gary Marshall
Executive Vice President & CFO John Day
President, Technology Solutions Group Robert Faletra
President, Business Technology Group Adam K. Marder
President, Healthcare Group Vicki Masseria
President, Specialized Technologies Group Regina Starr Ridley
President, Electronics Group Steve Weitzner
Senior Vice President, Business Development Vittoria Borazio
Senior Vice President, Global Sales & Marketing Bill Howard
Senior Vice President, Human Resources & Communications Leah Landro
Vice President & General Counsel Sandra Grayson
Vice President, Creative Technologies Philip Chapnick

CMP United Business Media

GamaNetwork

WWW.GAMANETWORK.COM

SAYS YOU

A FORUM FOR YOUR POINT OF VIEW. GIVE US YOUR FEEDBACK...

Game Engines, or Are They?

I object to Andrew Kirmse and Daniel Sanchez-Crespo's classification of NetImmerse and Alchemy as "game engines" in "Test Drive: On the Open Road with Two of Today's Most Powerful Game Engines" (December, 2001). They don't qualify as engines at all; they're component sets.

Anonymous
via e-mail

DANIEL SANCHEZ-CRESPO RESPONDS: *The term "game engine" is pretty slippery. Traditionally, it has referred to "closed solutions," which allowed the developer to concentrate on content creation. In this respect, NetImmerse and Alchemy should be better called "game development toolkits." Both Andrew and I took care that this idea was clearly stated in our reviews.*

Still, this semantic precision is a double-edged sword. Being "traditional" engines, Quake & Unreal should allow teams to concentrate on content, right? Still, when Valve used the Quake 2 engine to create HALF-LIFE, some components were reworked or written from scratch. Does that make Quake 2 less of an engine? Now, consider HIRED GUNS, a game built on top of Unreal. What's so interesting about it? Well, it's a real-time strategy game, clearly not what the people at Epic had in mind when they coded their software. The whole interface was replaced, and I can guess lots of AI/logic code needed some major reworking. Should we change the naming of those packages in that case?

Most teams working on classical game engines are in fact using a toolkit approach: analyzing the available components and reworking those that need it. Should we totally drop the "engine" term if even the most representative products violate the definition? Lots of precision can make it impossible to classify items into groups, as each item has specific features that make it unique. Being practical (and, yes, adding some imprecision), we can consider Quake and NetImmerse members of a same family which, for historical reasons, we can call engines. You will always have closed engines, which will offer a faster time to market, and toolkits, which give us greater flexibility. In the end, all these products are nothing but close relatives, so the incurred imprecision in the naming is, in my opinion, justified.

Kudos for "The Inner Product"

I have just read Jon Blow's first "Inner Product" column ("Mipmapping, Part 1," December 2001). I am currently on a game project and the information he has supplied is more than enough to get my team thinking on how we could implement a better mipmapping algorithm.

Steve Marth
via e-mail

Don't Forget to Gamma Correct

I'm so glad to see that Jonathan Blow's "Inner Product" column has picked up the technical torch at *Game Developer* ("Mipmapping, Part 1," December 2001).

Besides the ringing mentioned in the article, another effect usually ignored by game programmers (and almost everyone else) is the effect of gamma correction (or lack thereof) on PC monitors. Since I didn't see this effect mentioned in the article, here's a brief rundown. Imagine your base texture is a checkerboard, 2×2, two black and two white squares. The pyramid derived from this texture is a 1×1 gray, usually stored as 0.5. However, the answer should really be more like 0.73, if you factor in gamma correction for CRT monitors (LCD monitors screw up the equation usually, as their response is different).

Who cares? Well, in this case, if you use 0.5 then the object is far away and appears dim, and as it comes closer, it gets brighter. Not a huge deal, but it's so easy to avoid if you're precomputing mipmaps using elaborate filters (such as the article describes). You might as well get gamma correct, too.

If you're smart, you do all this with at least 12 bits of precision per channel, to avoid banding (Jim Blinn talks about this precision problem with gamma conversion in his books).

Eric Haines
via e-mail

JONATHAN BLOW RESPONDS: *Actually, in an upcoming column, I talk about gamma correction. Rumor has it that the next chip design from A Major 3D Accelerator Maker has deep enough channels and a versatile enough RAMDAC or page copier that you can just keep the frame buffer in light-linear space and exponentiate everything after the whole frame is drawn. This has good connotations for lighting (in other words, it becomes basically free to actually do lighting at the proper falloff rate).*

Teaching Games

I enjoyed Celia Pearce's "Learning Curves: The Present and Future of Game Studies" (Soapbox, December 2001).

Six years ago I proposed a videogame programming course to Paloma College, a community college in San Marcos, Calif., which I have been teaching part-time for the past five years. We recently decided to expand our program by offering two new courses. The first will be offered this spring semester: "An Overview of the Videogame Industry." The second class to be started next year will be a game programming course. We will be offering a videogame specialist certificate, but hope to expand it to an A.A. degree in videogame programming.

Ed Magnin
via e-mail

Let us know what you think: send e-mail to editors@gdmag.com, or write to *Game Developer*, 600 Harrison St., San Francisco, CA 94107

INDUSTRY WATCH

THE BUZZ ABOUT THE GAME BIZ | *daniel huebner*

Christmas console launches. The month leading up to the all-important holiday period saw the start of the second round the ongoing console wars, as Xbox and Gamecube finally made their long-awaited debuts just days apart. While both new consoles initially seemed to sell out as quickly as stores could stock them, in the weeks following the launches it was difficult to decipher the numbers to conclude who actually had the better launch.

Nintendo claimed that sales of nearly 600,000 consoles in the first 15 days of Gamecube availability made the launch the most successful console debut ever. While Microsoft had just half as many units ready for launch and hadn't publicized its sales numbers at press time, the company claimed the most successful launch title for Xbox, asserting that Bungie's HALO was out-selling Nintendo's LUIGI'S MANSION. Both companies hoped to have more than 1 million consoles in consumers' hands by the end of the year.

Microsoft proclaimed HALO the best-selling console launch software for the holiday shopping season.

The other consoles. Sega was moving the final Dreamcasts out the door after cutting the price on remaining consoles from $79.95 to $49.95. New pricing has moved the discontinued machine at a brisk pace, pushing Dreamcast sales past the 10 million mark. Said Sega's Peter Moore of the too-late sales surge, "Ironically, we now wish we had more."

SUPER MONKEY BALL proved to be a popular Gamecube launch title.

Sega's post-Dreamcast recovery is moving along slowly, as the company posted a $169 million loss in the six months ended in September. Sales were down 18 percent from the previous year.

Sony set a new price for Playstation 2 at the end of November, but the new rate didn't extend to North America, at least for the duration of 2001. The 15 percent price reduction was credited to reduced production costs rather than as a response to console launches from competitors Nintendo and Microsoft.

Bleem emulator gives up. Bleem, the company that made Playstation emulators for PC, Macintosh, and Dreamcast, has shut its doors after protracted legal battles with Sony over copyright infringement. Sony first sued Bleem over its products in May 2000. Bleem counter-sued, claiming Sony was exercising an illegal monopoly over the videogame industry.

Nvidia replaces Enron on S&P 500. The sudden demise of energy firm Enron Corp. was good news for chip maker Nvidia. Standard & Poors announced at the end of November that Nvidia would replace Enron in the prestigious Standard & Poors 500 Composite Index.

Square CEO quits after poor showing by *Final Fantasy*. Game software maker Square announced that president and chief executive officer Hisashi Suzuki would resign after the company reported its worst-ever loss for the first half of its fiscal year due to a disappointing showing by *Final Fantasy: The Spirits Within*. Square reported a group net loss of $106.8 million for the six months ended September 30. Chief operating officer Yoichi Wada was scheduled to take the top position on December 1, while Suzuki remains as the chairman. The film generated box-office revenue of about $30 million in the U.S. market, well below the targeted $80 to $90 million, and interest among Japanese consumers has also been weak.

Konami group net profit plunges. Konami announced a drop of 78.3 percent in group net profit in the first half of the company's fiscal year. Most of of the deficit was attributed to profit shortfalls for the company's Yu-Gi-Oh card game. Group net profit dropped to $20.74 million from $94.64 million for the six months through September, despite total sales rising nearly 20 percent to $725.48 million. Growth in sales of its videogames business, however, could not offset a sharp decline in operating profit in its card game business.

Interplay reports few bright spots in third-quarter financials. Interplay reported net revenues of just $4.2 million for its fiscal third quarter, a drop of 87 percent from last year. The net loss for the period was $20.6 million, a disappointing result after reporting net income of $0.1 million in the same period last year. Most of the drop can be attributed to failing to ship games; Interplay didn't have any new titles in the third quarter and had shipped just seven for 2001 as of mid-December.

Interplay released a total of 26 titles in 2000.

UPCOMING EVENTS CALENDAR

D.I.C.E. SUMMIT

HARD ROCK HOTEL
Las Vegas, Nev.
February 28–March 1, 2002
Cost: variable
www.interactive.org

GAME DEVELOPERS CONFERENCE 2002

SAN JOSE CONVENTION CENTER
San Jose, Calif.
March 19–23, 2002
Cost: $195–$1,950 (early-bird discounts available)
www.gdconf.com

Newtek's Lightwave 3D 7b

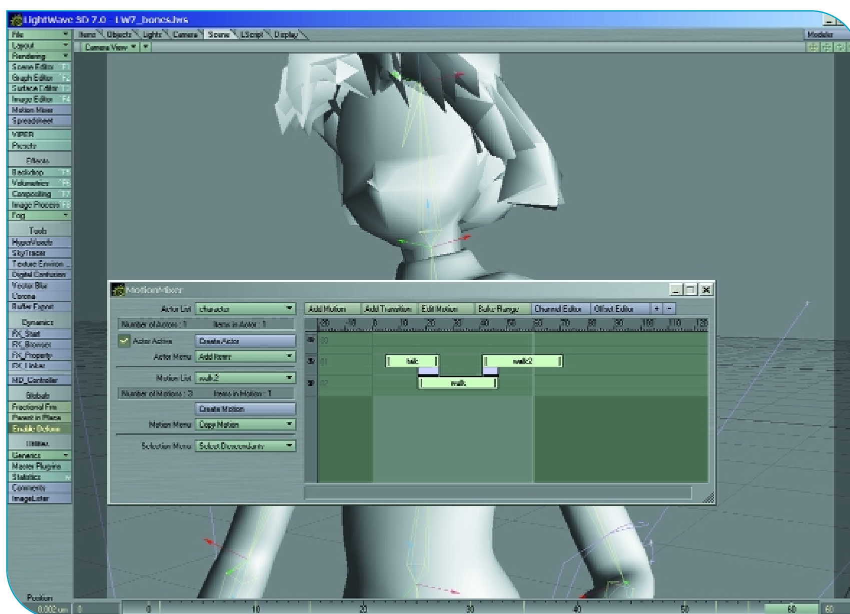
by sergio rosas

Newtek's latest release of its 3D modeling and animation package is crammed full of goodness. So what do you get for the upgrade price? You get some new features, some great enhancements to the established ones, and an occasional simple but elegant refinement.

Motion Mixer. Finally, Lightwave 3D gets its very own nonlinear animation system. Lightwave users have been waiting for a long time, and it's finally here: Motion Mixer. With this tool, you can define a character and then build a library of its motions. Once you've identified the motions (such as a run cycle and a walk cycle) you can edit them together and easily blend from one to another. Additionally, Motion Mixer allows you to set pre- and post-behaviors for motions (such as repeat and oscillate), scale, load, and save entire hierarchies of motions. It also has a nice motion mapper tool that lets you load motions from a different character.

Motion Mixer is a great tool, but after I used it for a little while, it left me wanting more. Unfortunately, Motion Mixer restricts its motions to a minimum of five frames. This means that you can't work with simple poses. Motion Mixer also lacks the ability to load in standard motion capture files. Nevertheless, if you still do character animation in Lightwave, this feature alone is worth the price of the upgrade.

Spreadsheet editor. Another great new feature in Lightwave is the spreadsheet editor. This view looks similar to the scene editor, except that you can expand it spreadsheet-style with tons of item properties. All of these properties can be selected en masse and tweaked. You can reset them all to a new value or add/subtract/multiply an offset to every one.

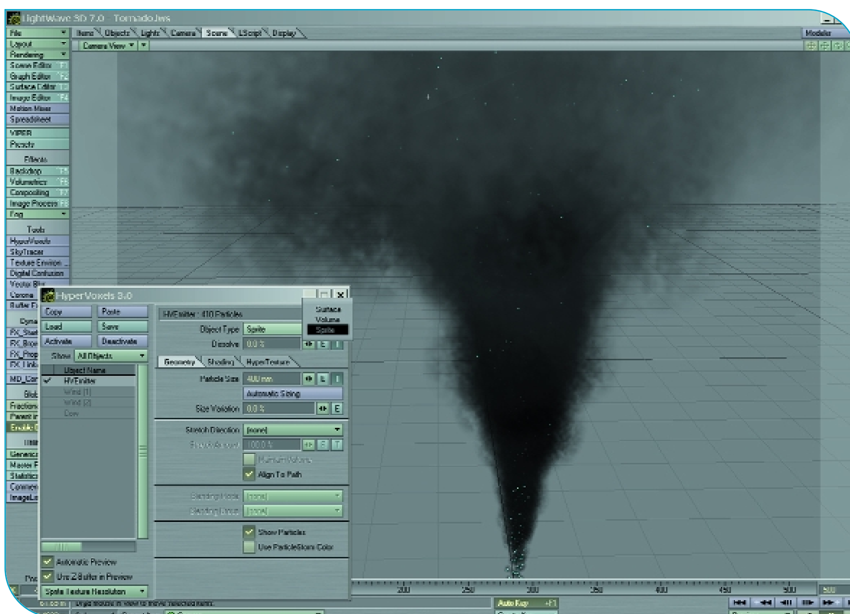


LIGHTWAVE 7B. Motion Mixer is Lightwave's long-awaited new nonlinear animation system.

You can make a massive change, preview it in real time, and if you like it, you can apply it to the scene. It also has a timeline, just like the scene editor, where you can move and size motions. You can make custom workspaces and save them with your scene. You can sort by any property and use its powerful filter for the items list. It took me all of two seconds to figure out and love this feature. I really wanted this feature when I had a scene with 78 lights that needed lens flares. Just for fun, I loaded that old scene up and was sorely disappointed to find that the speadsheet editor doesn't do lens flare properties. In short, the spreadsheet editor adds some extremely useful functionality, but you still might have to resort to a text editor every once in a while.

Sprites. Some might not consider this feature to be a big deal, but coming from game land, sprite-based particles are a huge deal to me. Both hypervoxels and volumetric lights have sprite options now. There's nothing to it, just click the sprite checkbox and let Lightwave take over. It renders the sprite version of your hypervoxel internally and assigns it to each particle. (Of course, you can also assign your own texture if you want.) This is a very useful option for game developers. You can see the particle sprites animating in layout in real time. Even though it's not exactly the sprite that being rendered, it's great for tweaking timing. When rendered, both the voxel and volumetric light sprites look just fine, even when flying through them. And they render a lot faster than in pre-

SERGIO ROSAS | *Sergio is currently acting as lead artist for THIEF 3 at Ion Storm. He can be reached at srosas@ionstorm.com.*



LIGHTWAVE 7B. Sprite options for hypervoxels and volumetric lights make these viable effects for game animators.

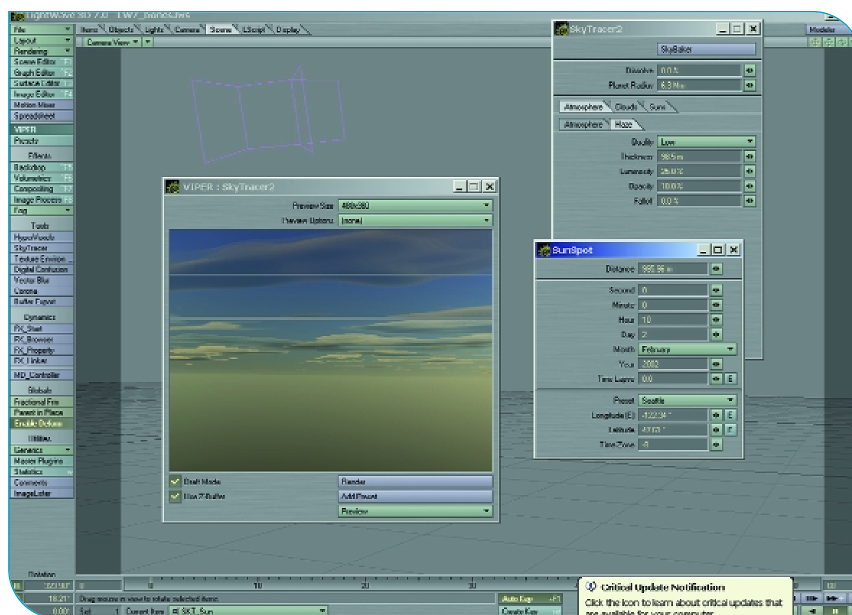
vious versions. This makes hypervoxels actually usable for me.

Rendering speed. Speaking of speed increases, Newtek added quite a few with Lightwave 7b. Besides being able to use sprites to speed up hypervoxels, you can also bake a hypervoxel cloud to get great rendering speed increases. Radiosity got a new "backdrop only" setting that renders faster even if it doesn't look quite as good. Also, a new "shading noise reduction" global illumination option can make the low-end area light and radiosity settings look less grainy while rendering much faster than the higher settings.

Enhanced particle system. Lightwave comes with an integrated particle system, and (although it might not be as good as the third-party systems) it's getting much better. With this release, the particle engine got interparticle collisions, particle respawning, and collision spawning. This means that you can simulate rain — each particle spawns little splash particles when it hits the floor. You're now able to attach an object or hierarchy to a particle, so you can make giant meteors crashing into each other and breaking into tiny pieces. Particle emitters also got a bunch of new nozzle types, such as sphere, cone, and object vertices. Newtek also added a bunch of new wind types, such as turbulence, vortex, and explosion. The wind types are powerful and have a great visual representation that makes them a snap to use. Everything about the internal particle system feels a lot more integrated than in previous versions — they even interact with motion designer soft bodies.

Graph editor improvements. With every new version of Lightwave, the layout graph/curve editor gets better and better. This time Newtek added an OpenGL interface, which the company claims is able to do hundreds of curves at once. It does feel much smoother and faster than it ever did before. Another new feature that's a great time-saver, the graph editor will now open with the curve for the current item selected. Unfortunately it doesn't open all of the curves if you have multiple items selected. Some other enhancements include match footprint, key reduction, curve filtering, and key bins.

Modeler enhancements. Compared to the Layout interface, the Modeler got very few new features. Newtek added a rounded box primitive and a more useful curve tool. The new face collapse command lets you select adjacent polygons and collapse them all to a single point. This is a real time-saver for optimizing models. You can also now airbrush between morphs. (Neat, but it's no Artisan.) Additionally,



LIGHTWAVE 7B includes Sky Tracer, a powerful and elegant feature for creating atmospheric effects.

being able to save a single layer as an object and to flatten all layers will come in handy to game animators.

LScript Commander. This new tool keeps a list of all the behind-the-scenes commands that are executed when you're clicking around the Layout interface. You can create macros easily by copying and pasting selected commands into the work area and hitting the Execute button. You can also install your own macro button from there. LScript Commander is not as slick its Maya or 3DS Max counterparts, but it's a start.

Sky Tracer 2.0. Sky Tracer got a facelift for Lightwave 7b. It's now fully integrated into Viper and seems easier than ever to use. It has a great Suns feature, which lets you type in the month, day, year, geographical location, and time, then it instantly pops out a great looking sky. It also has a Baker feature that easily exports a skybox — another old plug-in made usable for game developers.

Toon shading. For those of us who like stylized renders, Lightwave 7b ships with BESM (Big Eyes Small Mouth), a great shader for cartoon rendering. It has plenty of great options such as unlimited zones of shading and variable blending between zones. Each zone has opacity, brightness, and saturation settings. Specularity options, gradient overlays, edge options, and much more make this my favorite toon shader.

To Buy or Not to Buy?

Lightwave 7b has some very useful and long-awaited new animation features that might seem a little rough around the edges to some. Those accustomed to other character animation packages will probably not be tempted to make the switch. Still, these new features, along with all the great refinements made to existing features, are a must-have for veteran Lightwave animators. In the realm of special effects, Lightwave's hypervoxels are top notch and continue to refine. The addition of sprites makes Lightwave game developer friendly. For the low-polygon modelers, this version of Modeler has few new features, but it continues to be the best polygon modeler for the money. If you are doing UV mapping in Lightwave, you should upgrade to 7b. For anyone doing cinematics, or for rogue contract artists, I would make this my tool of choice. To cut a long story short, the "b" in Lightwave 7b stands for "bang for buck."

LIGHTWAVE 7b ★★★★

STATS

NEWTEK
5131 Beckwith Blvd.
San Antonio, TX 78249
(210) 370-8000
www.newtek.com

PRICE
$2,495 (upgrade price is $495)

SYSTEM REQUIREMENTS
Windows: Windows 98/ME/2000 (Service Pack 2)/NT 4 (Service Pack 6a), TCP/IP network protocol installed, 128MB RAM. Macintosh: PowerPC Processor (G3 or higher recommended), Mac OS 9 or Mac OS X (recommended), 384MB RAM for Mac OS 9, 128MB RAM for Mac OS X. All systems require 32MB available hard drive space, CD-ROM for installation, and a minimum screen resolution of 800×600 pixels.

PROS

1. Motion Mixer nonlinear animation tool
2. New spreadsheet editor
3. Sprite-based hypervoxels and volumetric lights

CONS

1. New features (Motion Mixer, spread sheet editor, vertex paint) seem disconnected and inconsistent with the rest of the package
2. Motion Mixer nonlinear animation tool can't use single-frame poses
3. No motion capture import

ACCUREV 3.0.1

by michael saladino

Revision control software is the safety net that allows multiple programmers to work together on the same code base without trampling on each other. Most systems fall into one of two camps: either a low-cost solution, such as Microsoft's Visual SourceSafe, or a massive system requiring a team of experts to keep it running, such as Rational's ClearCase. Accurev is trying to carve a niche for itself between these two extremes.

I was able to get my trial copy up and running within minutes of downloading it. The server setup was simple and painless. I had slightly more difficulty when working with the client side, but that's undoubtedly because the client is where the greatest amount of functionality is located. My test case was to take my current game project and copy it to Accurev. More than 2,000 files were uploaded into the system in a matter of minutes, my first indication that Accurev is extremely fast. Soon after this, I was able to get complete source for my client machine with all the basics, including difference, history, and checkout control. My next task was to time basic operations, and what I found was impressive. From checkouts to history differences, Accurev performed standard functions many times faster than the Visual SourceSafe equivalents.

However, where Accurev really shines is in its solutions to the subtle "gotchas" that often slow production when one uses low-budget software. One of the classics is when a programmer checks in a large group of files at the same time that another programmer is getting the latest code. The person getting code will only receive part of the total check-in, which in most cases will render the build unable to be compiled. Accurev disallows this by keeping transactions such as large check-ins from applying publicly until it is complete.

A more severe test that I ran was to begin a check-in of multiple files and then disrupt the transaction by pulling the plug on the Accurev server (a little extreme, but I'm testing reliability here). I know from experience that losing power during a Visual SourceSafe transaction can be a dangerous event, resulting in corruption of the database. Accurev, however, was able to survive nicely. When I restarted, everything was

functioning and the check-in process could be restarted on the client with no loss of data.

My only serious concern with Accurev is its interface design. While certainly easier to use than industry standards such as ClearCase, it still lacked the simplicity of Visual SourceSafe. The supposedly simple matter of bringing new files into the database was something that kept me stumped for far too long. Not surprisingly, I found it easier to integrate SourceSafe with Microsoft's Visual Studio. And while a cursory inspection of the software might remind you of SourceSafe's look and feel, its internals are definitely different and do require time to learn. I found that by the end of a couple hours of testing, I was beginning to feel more comfortable.

Accurev 3.0.1 is available for Windows 95/98/M/NT 4.0/2000/XP and an impressive host of other, more obscure platforms. The evaluation version is not time-limited, but is limited to two users. Accurev will provide interested parties with a price quote for extended licenses.

For my next project, I will certainly experiment more with Accurev to determine whether it would be a worthwhile change. From my initial tests, it's a promising new option for software developers no matter what the size of the project.

✶✶✶✶ | Accurev 3.0.1 | Accurev
www.accurev.com

Michael Saladino is senior programmer at Presto Studios in San Diego.

METRIC HALO'S SPECTRA FOO

by gene porfido

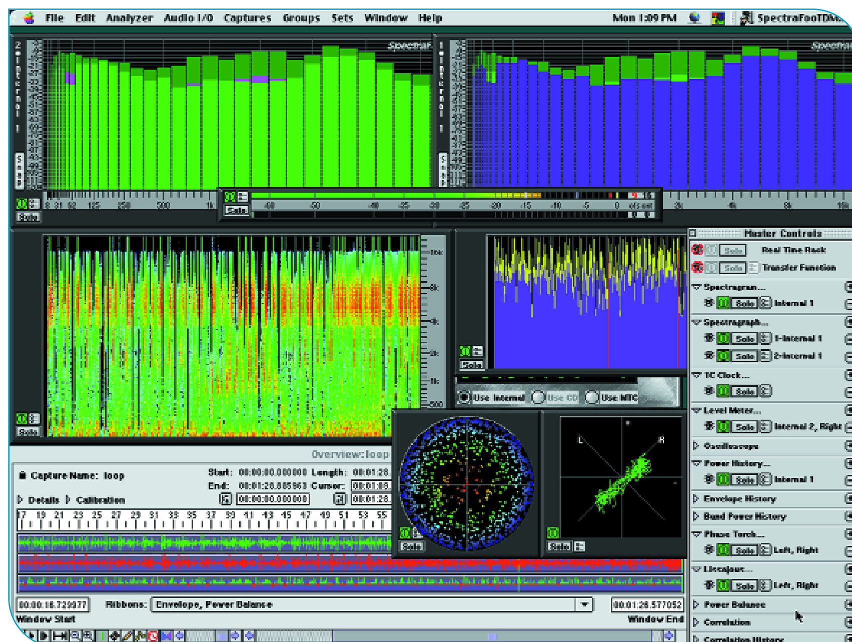
Metric Halo appeared on the audio radar a few years ago with a dazzling new program called Spectra Foo. "Spectra who?" you ask. That's Spectra Foo and it's an RTAS and MAS plug-in with an incredible array of functions that make it as easy to remember as its name.

Since its introduction, Spectra Foo has become the Macintosh standard for every imaginable audio-monitoring or test-job function one could dream of, and it has matured well in later versions. Some call it the Swiss Army knife of audio applications. And for professionals from mastering engineers to sound designers, it's like having a toolbox full of ultra-accurate meters, an expensive oscilloscope, and waveform displays built into your Mac.

There are two versions of "Foo," as it is commonly called, including the $400 standard version and the $800 Spectra Foo Complete. Each version is capable of high-resolution metering and measuring spectral analysis, phase correlation, waveforms, power, envelope, and spectral histories, as well as a number of highly configurable input/output configurations for matching or comparing audio signals.

While the standard version of Foo is extensive on its own, the complete version adds quite a few top-end features that augment its already impressive functions. There's a world-class signal generator that performs Pink and White noise, Burst, FFT, and direct-to-audio signal generation at 24 bits and with up to nine simultaneous sine wave sweeps. A Transfer function measurement system can handle equipment and test verification, acoustic correction, and time-alignment for studios or labs, and takes direct measurements, including frequency and phase, of acoustic and equipment signals. Add the Capture and Analysis system to record and open sound files directly into Foo, and you can begin to see how incredibly intricate this software package is.

The program runs well on a PowerPC 604e chip at 180MHz, but I'd recommend a G3 or G4 for heavy-duty work. It's very easy to set up as a plug-in inside of Pro Tools or Digital Performer (RTAS and MAS audio systems, respectively), but can also be opened as a stand-alone application. Setting up Foo to take audio inputs from your Mac I/O, CD, or other input source is no harder than it would be with your favorite DAW. With sufficient processing power, multiple instances can be opened simultaneously to monitor a mix or any number of individual tracks within your DAW program. Window Sets, which are customizable and can be assigned to a hotkey for quick access, offer a snapshot feature of every parameter for instant recall or retrieval as a preset. And a new Link feature keeps all of your parameters from any set of instruments and meters associated with as many groups of instruments as needed.



SPECTRA FOO gives sound engineers a stable and feature-laden toolbox for analyzing digital audio.

Spectra Foo is a name worth remembering if audio has any importance in your life. Whether your mastering the next Pink Floyd record or just making sure your game's sound effects are panned and in phase, this is an application that will come in handy in more ways than you'll ever imagine. Count on superlative support from Metric Halo's Dan Metivier, along with the company's incredible (and growing) lineup of exceptional audio software and hardware, and you'll soon be singing the praise yourself. This is what great software is all about.

★★★★★ | Spectra Foo | Metric Halo
www.mhlabs.com

Gene Porfido has been making music for over 25 years and game sound design for nearly a decade. He is currently an independent sound designer and composer in San Francisco.

MUSICLAB'S RHYTHM 'N' CHORDS 2

by todd m. fay

Rhythm 'n' Chords is a plug-in for Cakewalk and Steinberg's Cubase series sequencers that takes advantage of proprietary performance-modeling technology. Performance modeling focuses on emulating the nuances of human instrumentalists through computer technology. Rhythm 'n' Chords models rhythm guitar performances specifically. Creating realistic guitar parts by hand via MIDI can be an extremely time consuming activity. Rhythm 'n' Chords alleviates MIDI musicians from having to stress over those all-important guitar parts.

Musiclab delivers the plug-in directly from their web site, and here are four different versions for download. Rhythm 'n' Chords Lite is a free download available as a demo. Rhythm 'n' Chords 2 Standard, Rhythm 'n' Chords Pro, and Rhythm 'n' Chords Pro Gold all support an increasing number of features. Musiclab also offers libraries of strumming and picking patterns programmed by a team of engineers and musicians for use with the Rhythm 'n' Chords plug-in, but most of the libraries are included with the Gold version of Rhythm 'n' Chords. There are currently 18 different libraries featuring popular styles such as jazz, oldies, blues, rock, world, and funk. These libraries provide musicians with starting points for creating rhythm guitar parts in a particular style.

For those of you seeking the ultimate in guitar performances, there is still only one tried and tested method for capturing truly inspiring work — that is, of course, recording a guitarist. For the musician working on a budget or composer working on a preproduction version or a recording, however, this plug-in can be a real timesaver. After using Rhythm 'n' Chords to lay the groundwork for the rhythm guitar parts, the arranger is free to further "humanize" the content. The plug-in makes it possible to create hours' worth of work in mere minutes.

RHYTHM 'N' CHORDS offers an interface that will be familiar to musicians.

With more and more games taking advantage of interactive MIDI-based music production, a tool that can aid in the production of realistic musical performances via MIDI is a welcome addition to any game audio professional collection. This plug-in can help raise the production value of a piece while still allowing for the advantageous small file size associated with MIDI.

Knowledge of the guitar helps, but is not required. The interface is similar to a jazz chart. You select chords from a list of supplied voicings and enter them into the measure. Then you either select a rhythm pattern from a library or use one that's been custom created. The arpeggiation time of the strum, chord voicing, and tone can all be adjusted quickly via an easy-to-use interface. Chord voicings, for example, are spelled out on a miniature graphical keyboard, so even those with no exposure to guitar playing can create the right harmonies. You can even control the amount of muting without having to switch to a different guitar patch. This is done using Rhythm 'n' Chords' virtual sliders. There is also a manual play feature that allows the plug-in to be used with a keyboard. This way, chords can be voiced by playing them on the keys while the plug-in plays through the user-selected rhythm patterns. This feature is especially useful for those who like to spend time with their hands on the keyboard and off the mouse.

Note that Rhythm 'n' Chords is only available for use in Cakewalk and Cubase products and only on the PC. Like everything else in computer music production, you win some and you lose some. Except in this case, players without access to this plug-in maybe losing out big-time. Unless you're a master guitar player with access to a MIDI guitar, I strongly recommend you pick up this little beauty right away.

★★★★ | Rhythm 'n' Chords 2
Musiclab | www.musiclab.com

Todd M. Fay (aka LAX) is an audio consultant and content creator working in and around the games industry. E-mail him at todd@lax-element.com

PROFILES

TALKING TO PEOPLE WHO MAKE A DIFFERENCE

Warren Spector's Sid Meier!

In the world of game development, Sid Meier is as close as one can get to being a man who needs no introduction. Since co-founding Microprose in 1982, Sid has designed and programmed dozens of games that have been heralded as nothing less than revolutionary, ingenious, and influential to all who follow in his footsteps. Sid lends his experienced hand to Firaxis Games as chairman and creative director of the recently released CIVILIZATION III and the upcoming SID MEIER'S SIM GOLF.

This month's questions were provided by Warren Spector, who has designed numerous critically acclaimed games for Origin Systems, Looking Glass Studios, and Ion Storm. Warren is currently executive producing Ion Storm's upcoming DEUX EX 2 and THIEF 3.

Sid Meier: Happiest when programming.

Warren Spector. **What are Sid Meier's inspirations? Do you play a lot of games? Do you look to the cultural zeitgeist? Do you specifically and consciously look outside the universe of games for fresh insights and ideas?**

Sid Meier. Most of my game ideas trace back to my childhood, to things that I became fascinated with at some point during my childhood. Pirates, airplanes, trains, history, and the Civil War were all interests of mine at one time or another.

WS. **How do you start the game design process? Do you typically have a moment of gameplay in mind? Or maybe a story or fictional context? Maybe a single game mechanic you think would be cool? A particular fantasy you want to allow players to experience or an overall experience you want them to have? A mood you want to evoke or a message you want to convey? Where does a design start for you?**

SM. In starting a design I focus on two key moments. The first time a player starts the game, he or she needs to be quickly drawn into the game. At the end of the game, the player should have a sense of having come a long way since the beginning to a satisfying conclusion and be tempted to play again.

WS. **How much documentation do you do before beginning to work? Are you a preplan-as-much-as-possible guy or a prototype-and-revise guy? I've always heard the latter, but I want details!**

SM. There's really no preplanning when we start a new game. We build the game using stuff we already know, with the idea that our players will already know this stuff too, and they'll be able to jump right in. Later we do research to add depth, create scenarios, and get the details right, but not until we have a fun game.

WS. **How much "real work" do you do these days, and how much of your time is spent conveying a vision to a team, or melding various team members' spins on the game into a seamless whole, or just dealing with team and studio management issues?**

SM. Actually, I enjoy programming and I don't enjoy management, so I'm generally the lead programmer on at least one project.

WS. **How do you explain your success? You've probably worked in a greater variety of genres than anyone else in this business — science fiction, historical sims, pirate games, espionage adventures. Do you think your greatest successes were driven by the appeal of a specific genre or fiction, or were there gameplay differences that made the difference, sales-wise?**

SM. I don't really know how to predict the success of a game. In hindsight, it might seem that doing CIVILIZATION was a no-brainer, but at the time it was a real departure for Microprose. At the time, strategy games were considered boring and complicated. I write games that I think I would like to play and hope there are some other people out there who will like them as well.

WS. **How tight is the link between genre and gameplay? In other words, can the same mechanics be applied to a sci-fi game as to a historical sim? Does genre dictate gameplay and game mechanics, or do the mechanics come first and then the genre?**

SM. We pick the game topic first and then worry about the mechanics. CIVILIZATION started out as a real-time game and switched to turn-based. PIRATES! was a combination of storytelling, adventure, and action. I tried three different approaches to the DINOSAUR game — turn-based, real-time strategy, and a card game approach — before finally giving up.

WS. **A lot of folks in my studio look at some elements of ALPHA CENTAURI in particular as a model for some of the things we hope to do in future immersive simulation games. Do you ever look at other people's games, regardless of genre or game style, and see some of your own ideas embodied in them? Conversely, do you ever ask yourself why more game developers don't adapt your ideas to their own work? In other words, how do you feel your games have influenced the development of games and gaming?**

SM. I think there is a continual sharing, borrowing, and building upon game ideas among the design community. As long as each game also introduces some new ideas and innovation, this is one of the strengths of our industry. Certainly the standardization of interfaces and controls has made games easier to play. I still love to play games. I hope other designers will continue to create great games so that I can play them, and occasionally borrow an idea or two.

Mathematical Growing Pains

This month, I'm going to talk about how to represent lines and planes, the sorts of linear entities that programmers manipulate all the time.

In high school I was taught that the equation $y = mx + b$ is a groovy way to represent a line in 2D. The equation is useful because m represents the slope and b is the y-intercept — that is, the line intersects the y-axis at $(0, b)$. This representation is good if you don't have a lot of higher math experience and you just want to draw a line on a piece of graph paper: b gives you a starting point, and m gives you the direction to go from there.

Years passed, until one day I was programming some pretty advanced 2D games; by then I had used $y = mx + b$ for visualization so often that I thought of it as the primary way to talk about lines. So I tried to make systems that represented lines with two floating-point numbers, m and b.

But what happens when a line is vertical? Its slope is undefined. In that situation, in high school, you'd just write $x = k$, which seemed simple enough. But with games, you have to think about more complex situations, like lines that are smoothly rotating from frame to frame. And you're writing code that uses limited-precision numbers, so your computations become numerically ill-conditioned when the lines are steep, because m is such a huge number. To fix this problem, you put a bunch of `if` statements into your code to change the computation based on what neighborhood the slope is in. That's not desirable from a software engineering standpoint, and the computational discontinuities (these happen as your parameters cross from one `if` scenario to another) may cause subtle but disturbing things to occur. See the pseudocode in Listing 1 for an example.

These problems go away when you make a simple mental adjustment and use $ax + by + c = 0$ as your line equation. This is like the slope-intercept equation, but before a division has taken place; if you divide $ax + by + c = 0$ by b (the b from this equation, not the b we were talking about before), you get the slope-intercept form. The slope and intercept shoot toward infinity when b is near 0, meaning the line is vertical. So $ax + by + c = 0$ is more robust because it doesn't divide by b.

As a bonus, the surface normal of the line is (a, b), and the distance from the line to the origin is c. You can easily read these features out of the equation, and being a game developer, you're more likely to care about these things than the y-intercept. Though we now need three floating-point numbers to talk about our line, a, b, and c, that extra number buys convenience and software reliability. The software becomes more reliable because the precision of our computations is more isotropic. In other words, it doesn't matter so much what direction the line goes in.

To sum up, my learning of $y = mx + b$ as the way to talk about lines had impacted my effectiveness in making games; the alternate representation removed those blockades.

Extending Lines to 3D

After a while, I'd made enough 2D games and decided to try 3D. When I first tried to formulate line equations in 3D, I got confused. In 2D, $ax + by + c = 0$ had been the best thing since sliced bread, so clearly I wanted to extend that equation to 3D. The obvious candidate is $ax + by + cz + d = 0$. I knew from reading books that this was the equation for a plane. Extending my line equation to 3D requires adding z in somehow, right? How else could I possibly add a z that would make any sense?

The problem is that $ax + by + c = 0$, which I'd thought was an enlightened way of representing a line, is not a line equation at all — and neither is $y = mx + b$, for that matter. It's a plane equation, and it only worked because lines and hyperplanes in 2D are the same thing (where my temporary definition of a hyperplane is "that which divides space into two halves").

There is an equation that works for all lines regardless of the space's dimension. It is $L = p_0 + vt$, where L represents the set of points comprising the line, p_0 is an arbitrary point known to lie on the

JONATHAN BLOW | *Jonathan is a game technology consultant living in San Francisco. Film that influenced this article:* Mulholland Drive, *the GREATEST FILM EVER. They can pretty much stop making movies now. It's time to work on games. Jon's e-mail address is jon@bolt-action.com.*

line, v is the direction vector that the line travels in, and t is the time parameter. When we get used to thinking about lines this way, we build up intuition that is valid no matter how many dimensions we're dealing with. We say that this is the parametric form of the line, as varying the parameter t will give you every point in L. If n is the dimensionality of your space, then this equation requires $2n$ numbers' worth of storage if you're being lackadaisical, or $2n - 1$ if you're being hardcore.

Simultaneous Equations?

So why is $ax + by$ + etc. the equation of a hyperplane and not a line? It's because it takes n degrees of freedom (represented by the coordinate variables x, y, . . .) and, by binding them together with the equal sign, places one constraint on that system of variables. This linear constraint removes one dimension; it flattens the space in the direction of the gradient of the equation (this gradient is the same thing as the normal of the hyperplane). The resulting space has $n - 1$ dimensions: in 2D, you get a line; in 3D, a plane; and in 4D, you get a 3D hyperplane.

Suppose we didn't want to use the parametric form for a line in n dimensions. Instead, we could represent the line by starting with the full n-dimensional space and squashing it $n - 1$ times, because $n - (n - 1)$ is 1, the dimensionality of a line. We can do this using $n - 1$ linear equations simultaneously. Simultaneous linear equations are the same thing as a matrix. So we're storing an n by $n - 1$ matrix, which uses a lot of storage space, and furthermore, it's not guaranteed to behave nicely. Suppose two of our equations try to squish the space in the same direction. After the first equation acts, there's nothing left for the second one to do; so the second equation doesn't reduce the space by a dimension (in fact, it leaves it unchanged). After all our $n - 1$ squashings, the remaining entity will have one more dimension than we expected; instead of a line, it will be a 2D plane.

LISTING 1. An example of how a singularity in mathematical representation affects code.

```
struct Line {
    float slope, y_intercept;  // ´slope´ == m, ´y_intercept´ == b
    bool is_vertical;          // or else declare this, meaning the above are invalid
    float x_value;             // used only if the line is vertical.
};

bool intersect_with_vertical_line(Line *vertical, Line *non_vertical, float *x_result,
float *y_result) {
     *x_result = vertical->x_value;
     *y_result = non_vertical->slope * vertical->x_value + non_vertical->y_intercept;
     return true;
}

bool intersect_nonvertical_lines(Line *line_1, Line *line_2, float *x_result, float
*y_result) {
     // Hope this denominator is not small.
     *x_result = (line_2->slope - line_1->slope) / (line_2->y_intercept - line_1-
>y_intercept);
     // Choice of line_1 below is arbitrary, hope we´re well-conditioned.
     *y_result = line_1->slope * (*x_result) + line_1->y_intercept;
     return true;
}

bool intersect_lines(Line *line_1, Line *line_2, float *x_result, float *y_result) {
    if (line_1->is_vertical) {
        if (line_2->is_vertical) return false;
        return intersect_with_vertical_line(line_1, line_2, x_result, y_result);
    }

    if (line_2->is_vertical) {
        return intersect_with_vertical_line(line_2, line_1, x_result, y_result);
    }

    return intersect_nonvertical_lines(line_1, line_2, x_result, y_result);
}
```

We then need to break out some advanced linear algebra to deal with the situation. Naive game programmer code, just consisting of a big hand-derived vector equation worked out on paper, will end up dividing by 0 somewhere and freaking out. More experienced programmers might use a matrix equation, but black-box matrix methods get screwy too; we end up with a situation where the determinant of a matrix is 0 and we want to invert it. The matrix has no inverse. Badly written code tries to invert it anyway, and thus produces inaccurate results or NaNs. Better matrix code takes stock of the situation with an `if` statement and, if the determinant is within some epsilon of 0, reduces the dimensionality of the matrix and solves a reduced-dimension problem. But picking suitable epsilons is not easy, and numerical discontinuities are introduced by the `if` statement.

All this should sound familiar from an engineering standpoint — it's the kind of thing we were doing with $y = mx + b$ when the line became vertical, and all the same problems arise. Cases of determinant 0 are often called "degenerate," but I think they are quite natural and inability to deal with them indicates weak methodology.

Imagine that you have three different planes, all passing through the origin,

rotating freely in 3D. You want to find the intersection of those planes. Most of the time, they intersect in a point; but if two of the planes coincide, then all three intersect in a line; and if all three coincide, the answer is a plane.

To solve this intersection problem using beginner's linear algebra, we write a matrix equation $p = A^{-1}d$ that finds the solution; but hard-coded into this equation is the assumption that the answer is a point. When the answer is not a point, A has determinant 0, so the equation is unsolvable. But what's the big deal? Sometimes planes are coplanar, just like sometimes lines are vertical. Why should that be a problem? The problem goes away when we stop treating matrices as black boxes that we want to invert, and instead start decomposing them and looking at their intrinsic properties. The QR and singular value decompositions become useful to us at this point.

Common Mathematical Misconceptions

I started this article with the question of how to represent a line. As 3D programmers we get past these problems early on, if only because we can't do lines in 3D otherwise. About the varying representations of a line, I want to develop an analogy: they are like other concepts that we work with from day to day, rooted in the core of our thinking, that are misleading in 3D and don't even work in higher dimensions. I'll now describe the biggest ones, the axis of rotation and the cross product.

The Axis of Rotation

When learning 3D math, once we get past the inconvenience of Euler angles, we find that all rotations can be represented by an axis vector, about which we rotate, and an angle, representing the magnitude of the rotation. Perhaps we visualize a rotation as a wheel turning on an oriented axle.

The problem is that the whole concept of "axis of rotation" only works in 3D. In 2D, rotations occur around a central point, and maybe we think of a nonexistent axis sticking out of the plane to help us visualize this. But a much more reasonable way to think of rotations is to speak of the "plane of rotation" rather than the axis. In n dimensions, any rotation occurs within a two-dimensional plane, and the object rotates "around" however many dimensions are left in the space. In 2D space, you rotate around a zero-dimensional subspace, a central point. In 3D, you rotate around a one-dimensional vector subspace. In 4D, you rotate around a two-dimensional planar subspace. (In 3D, the surface normal of the plane of rotation is the axis vector we are used to thinking about. In higher dimensions, using this definition of rotation, it's no longer true that you can reproduce an arbitrary orientation with only one rotation.)

I tend to think of rotation as "the thing that binds together any two dimensions of our space." In 3D, we have three canonical planes of rotation: the xy plane, which binds things that leave x to entering y, and likewise for yz and xz planes. Any rotation occurs within a 2D plane that is a linear combination of these three canonical planes. In 4D, there are six such canonical planes.

David Hestenes (see For More Information) uses a different terminology from what I use here; he speaks of "simple rotations," which occur within two planes, and "arbitrary rotations," which can reproduce any orientation in the space. I find this terminology unappealing, since in some higher-dimension spaces, such as 4D, a non-simple "rotation" may have no eigenspace — that is, no "axis" of nonzero dimension around which the rotation pivots. So I find it hard to visualize the thing he calls a "rotation." But your mileage may vary.

The Cross Product

The cross product is a fundamental piece of 3D math that we use all the time. But we were taught incorrectly what the cross product is and how it works, with the result that we often use it improperly, in subtle ways.

We are usually taught only about the cross product in 3D. But what is the cross product in four dimensions and higher? Does the concept even make sense? Because two linearly independent vectors determine a 2D plane, it is possible for us to interpret the results of the cross product in n dimensions as the subspaces we were rotating around just a few paragraphs ago: in 2D, the result is a scalar; in 3D, a vector; and in 4D, a 2D-planar thing.

Following this scheme, when we take the cross product in 3D, we think of it as returning a vector result. Unfortunately, this result is wrong, and we see this in a few places. A prominent symptom is that "surface normal vectors" can't be transformed in the same way that plain vanilla vectors can; if you are transforming vectors by some transformation T, you need to transform normals by $(T^t)^{-1}$. Beginning 3D programmers may not run into this problem, because if T is just a rotation, its inverse is equal to its transpose: $(T^t)^{-1} = T$.

This difference in transformations is necessary because the cross product is weird. We are providing two vectors as arguments of the cross product, and those vectors determine a 2-plane if they are not colinear. But the cross product implicitly returns to us the dual of that plane, its normal vector. So we think we're talking about a vector, but we're really talking about a plane through the origin. The plane occupies whichever two dimensions its normal vector does not; because of this, transformations can affect the plane in ways that we would not see if we considered its normal vector in isolation.

To ensure that we always pick the right transformation, we can say that the output of the cross product is a thing called a form, which one might think of as a transposed vector. The form interacts with matrices in all the ways you'd expect a row vector to behave. Smart physicists have been dealing with the differences between point-like and plane-like things for a long time; eventually someone invented Einstein index notation, which

helps demystify things. Jim Blinn (see For More Information) wrote two articles that discuss the Einstein notation from a graphics programmer's point of view. But this whole tensor algebra approach gets pretty complicated, so some new-school physicists are evangelizing Clifford algebra (also known as geometric algebra) as a method of simplification.

Clifford algebra defines the "wedge product" of two vectors in a way that is similar to the cross product, but it returns a nonvector result; that result is a plane-like thing called a bivector. You can take the wedge product of a bivector and another vector to get a volumetric trivector, and so on. The Clifford product of two vectors gives you a result containing both scalar and bivector parts; it is the dot product and cross product all wrapped together. This unification enables us to do things that make life easier, like dividing an equation by a vector or a plane.

In some references, the 2D version of the cross product is called the "perp-dot product" (see F. S. Hill's Graphics Gem in For More Information). Pertti Lounesto's book describes higher-dimension cross products that are different from the one I've mentioned here.

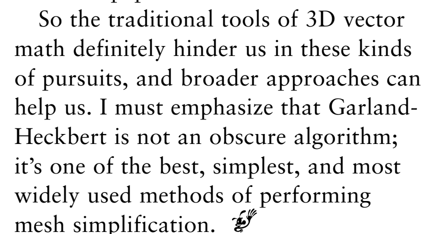
Why We Should Care About *N*-Dimensional Generality

Recently, to generate levels of detail for humanoid character meshes, I was implementing Garland-Heckbert Quadric Error Simplification (see For More Information). The basic version of this algorithm, which only takes mesh geometry into account, operates on 3D vectors; it uses 3D plane equations that are derived and evaluated using the cross product and the dot product. But to take vertex color and texture coordinates into account, we need to generalize the algorithm to higher dimensions.

We hit a wall when we try to move the algorithm to higher dimensions, because each face of our mesh imposes a two-dimensional constraint on the quadric error metric. When we're in three dimensions, this constraint can be represented as the hyperplane $ax + by + cz + d = 0$, which we're used to playing with. But when we go up to five dimensions (three spatial dimensions plus two texture coordinates per vertex), we no longer have such a tidy hyperplane equation to represent what's going on. Each face of the mesh defines a 2D plane, but now a 2D plane is just a small strand in the 5D space, so we need to represent it parametrically. This is exactly analogous to the way $ax + by + c = 0$ stopped working for lines when we jumped from 2D to 3D.

Another way of looking at the problem is this: in 3D we usually get a plane from two vectors by taking the cross product. But if we're not conversant in advanced linear algebra, it is unclear how to perform this process in 5D. So be sure to eat your multi-dimensional Wheaties.

In their paper, when the time comes to elevate above three dimensions, Garland and Heckbert shift gears away from the hyperplane approach and re-derive their algorithm differently. But if you start with an all-encompassing approach (such as Clifford algebra) from the beginning, the algorithm works no matter what dimension you deal with, and you never have to get confused or change your mode of thought. You also end up with a shorter derivation than that used in the Garland-Heckbert paper.



So the traditional tools of 3D vector math definitely hinder us in these kinds of pursuits, and broader approaches can help us. I must emphasize that Garland-Heckbert is not an obscure algorithm; it's one of the best, simplest, and most widely used methods of performing mesh simplification.

When we get used to thinking about lines and planes parametrically, we build up intuition that is valid no matter how many dimensions we're dealing with.

FOR MORE INFORMATION

Blinn, Jim. *Jim Blinn's Corner: Dirty Pixels.* Morgan Kaufmann, 1998.

Dorst, Leo. "GABLE: A Matlab Geometric Algebra Tutorial." http://carol.wins.uva.nl/~leo/clifford/gable.html

Garland, Michael, and Paul S. Heckbert. "Simplifying Surfaces with Color and Texture Using Quadric Error Metrics." *Proceedings of IEEE Visualization,* 1998. http://graphics.cs.uiuc.edu/~garland

Hestenes, David, and Garret Sobcyk. *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics.* Kluwer Academic Publishers, 1987.

Hill, Jr., F. S. "The Pleasures of 'Perp Dot' Products," in *Graphics Gems IV,* ed. Paul Heckbert. Academic Press, 1994.

Lounesto, Pertti. "Clifford Algebras and Spinors." *London Mathematical Society Lecture Note Series #239.* Cambridge University Press, 1997.

ACKNOWLEDGEMENTS

Thanks to Chris Hecker for redirecting the overly negative energy that originally permeated this article concept, and for some pointers regarding matrix decomposition.

Making Trees Work

How many times as an artist have you wished that you were working on TETRIS? Those small, colored blocks, slotting together — simple, Spartan, square. It may not be creative nirvana, but at least you know where you are with a square — four sides, four corners; you can't really go wrong. Other shapes are somewhat less forgiving, and those that relate to the real world are usually amongst the most awkward.

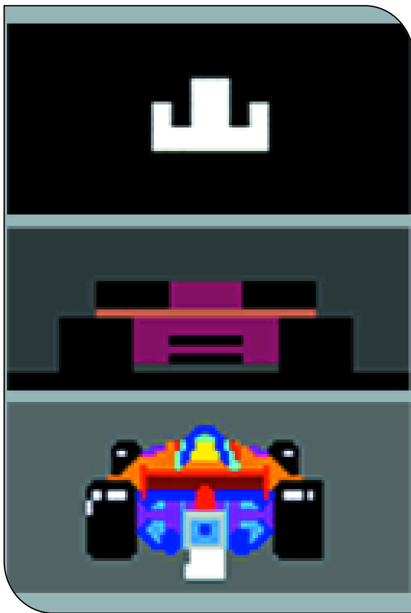


FIGURE 1. Formula 1 cars through the ages.

Those Were the Days

As we rocketed out of the 1970s and through the 1980s, toward the 16-bit wonders of the early 1990s, we left the minimalist angularity of PONG and the harsh, blocky SPACE INVADERS behind us, heading for the world of graphical beauty that glimmered on the horizon. Videogames slowly made the transition from the symbolic, where the sports car in our racing game looked like it had been drawn by my three-year-old brother, to the representational, where a Ferrari had at least a passing resemblance to the vehicle it was portraying. Figure 1 shows three transformations videogame cars have gone through over the ages.

Suddenly, games were no longer judged simply on how much fun they were; they also had to look fantastic. Teenagers across the globe found themselves huddling together around a single copy of a magazine they'd smuggled into school, breathing heavily, and letting out the occasional sigh of rapture. This time, however, it wasn't Marlena (21, enjoys skiing and topless volleyball) stoking the fires of their desire, it was a game. The centerfold had stepped aside; the pinup had given way to the pixel.

As with most things technological, the increase in graphical quality continued to grow exponentially. A palette of 256 colors soon became one of over 65,000. Screen resolutions grew and animation broke free from the confines of a simple loop. And then 3D hit us. Like a stampeding herd of hyperactive rhinoceroses, the 3D revolution trampled the sprite and all its flat friends under its huge, polygonal feet. The irresistible bulk of an industry ready to move on, backed by the widespread availability of the hardware necessary to do the job at an affordable price, moved games and their graphics to the next level.

And here we are. By the time I finish writing this article, there may well have been another wave of graphics cards released, capable of real-time volumetric refraction with a dedicated subprocessor specifically created to deal with dynamic nasal hair deformation. By the time my cup of coffee gets cold, the next generation of consoles may well be on the shelf, promising to draw things so fast that the only way to prevent your TV from melting is to immerse it periodically in liquid nitrogen. Today's game players want more than a yellow circle with a mouth being chased around a maze by ghosts. They want their games to be set in a world they can believe in.

Whether a game is trying to re-create a location that is real or it takes place in a world of fantasy, more often than not certain elements of the environment occur in both. Rocks are usually rocks, whether they're on an ancient Egyptian battlefield or in the lair of the mighty demon Aarath'ak the Unhelpful. Whether you find yourself snowboarding in the Rockies, or racing through the valleys of the planet Sprag-Thurman VIII, a tree

HAYDEN DUVALL | *Hayden started work in 1987, creating airbrushed artwork for the games industry. Over the next eight years, Hayden continued as a freelance artist and lectured in psychology at Perth College in Scotland. Hayden now lives in Bristol, England, with his wife, Leah, and their four children, where he is lead artist at Confounding Factor.*

(excepting certain localizing factors) is basically a tree.

I Hate Trees

From the outset, we have a few problems when it comes to trees. First, just like most things that nature creates for us, trees were never designed to be modeled economically in three dimensions. The whole branching concept is

FIGURE 2. Two different species of tree.

death to even the most robust polygon budget, and that's before we get to the leaves. Second, besides the fact that each specific species of tree is distinct from every other species, every single tree is always, in itself, unique. And finally, a tree is very rarely found in isolation; more often than not it will be in a group, and these groups (clump, copse, woods, or forest), are going to be outside, which is usually quite a large place to fill. Figure 2 shows the differences between two different species of tree.

There is a clever solution to this problem: simply set your game in a submarine. There are no trees in submarines. Or how about space? A space combat game will remove any need for trees. Just as long as you don't land on a deciduous planet, that is. However, chances are that neither of these scenarios will fit with your current project, and so a strategy to deal with trees will definitely come in handy.

Context

To begin with, it is worth looking at how trees fit in with your game design. This may seem like overkill, but it will ultimately affect the choices you make about the best way to approach building your trees.

As is often the case, some of the first questions to ask have to be: What kind of world are we building? Will players have full access to the whole landscape, or will they be limited to certain predefined areas? How much space will a level represent, and what percentage of this is likely to be exterior space? Will trees generally be used to beautify the background during bouts of tightly scripted action, or will they most likely be used to break the monotony of large, open plains? Do trees have a more important role within the gameplay? Will their placement be carefully integrated with the level design as cover to be used by players when they creep around, stealthily to avoid detection by the enemy? Do they provide a hiding place from which the Giant Spleen-Beast of Gaarg will charge at the end of the level?

Obviously, approaching any element of a game's visuals in isolation from its context is like painting a portrait over the telephone. Chances are, you're going to have to do it again. In this respect, the items of primary concern are: Are the trees themselves likely to be under close scrutiny by the player? What level of detail will strike the best balance between quality and speed?

It is entirely possible that your engine is able to apply some form of dynamic level-of-detail adaptation to geometry, which gives you extra breathing space when it comes to limiting your model's complexity. However, regardless of what hardware manufacturers around the world would have us believe, you and I both know that unless your engine draws its power from the Dark Prince himself, economical design is what we're after.

What Makes a Tree?

Before we set about building one, it's worth having a quick look at what makes a tree. Chances are, you're pretty familiar with the basic tree. Bear with me, however, while I break it down into a few basic elements:

Roots. Depending on the species, roots are generally underground, but particularly with larger, older trees, roots have a significant presence above ground.

Trunk. This fairly straightforward central mass of the tree typically starts fat, ends thin.

Branches. Mainly quite random, but some species, such as firs, have surprisingly uniform (fractal) branching structures.

Bark. There is enormous diversity from species to species.

Leaves/needles. Again, these are hugely diverse in shape, as well as relative size, when compared to the parent plant.

So that, more or less, is a tree. No problem. Couldn't be easier. Familiarity with the subject when trying to re-create it, whether in paint, plaster, or polygons, is bound to make the process easier, isn't it? Unfortunately, a faithful re-creation of something as complex as a tree will, by the time you've scattered trees across your level, most likely have your engine cowering in a corner, begging for mercy. In this instance, as is often the case when considering graphics for a game, achieving realism will involve a certain element of stylization and a level of approximation that fits within the restrictions of your particular project.

Cheap and Cheerful

Before you create a single vertex, it is helpful to divide the trees up into two groups: generic, or filler trees, and those that will have a more prominent place within a level, the feature trees (for want of a better name). A filler tree, as the name suggests, is most likely used in numbers to break up empty background space, and to create a more attractive, detailed exterior. The feature tree is generally the setting for some important action, whether gameplay or cutscene, and as such the player spends more time looking at it. Because the feature tree is a more central part of any scene you are creating, its level of detail, while dependent on engine limitations, is considerably higher than that of the filler tree.

FIGURE 3. Asymmetrical trees with rotation.

It is unlikely that you have either the time or the resources to create a vast array of individual trees that mimic their uniqueness in the real world. It is important, then, to design your filler trees to work well as a set that can be distributed as well as possible to create the illusion of variety. Scale variations and rotation (as long as the trees are not symmetrical) can be mixed with vertex coloring as well as light and shadow variations across a landscape to increase the apparent number of different trees that have been created. Figure 3 shows three asymmetrical trees with rotation.

Once you've chosen your basic species of tree (and here I mean, fir, palm, deciduous, and such, not *Chamaecyparis lawsonia*), you can create the object that will form the trunk. For a filler tree, this should have as simple a shape as possible (a triangular cross-section should be perfectly adequate in most cases), with the number of branches kept to a minimum and no secondary branches needed. Most of the work for this kind of tree can be done in texture. Detailed branch and leaf structures can only be represented economically in texture, and this is especially true in the case of the filler tree.

Foliage

The extreme economy of the past, with two crossed polygons displaying a complete tree in texture, may be some way behind us. However, using a combination of solid geometry for the trunk and major branches and some crossed planes representing the foliage can provide an adequate compromise.

Creating a useful texture for this part of the tree is of particular importance, and the following are four methods that are worth considering.

Hand Painting

Hand painting is useful if your visuals are adopting a particular style, and it's especially worth considering if you plan to work at a resolution of 128 dpi or lower. You can create leaves and branches in most decent paint packages and use features such as cloning and image hoses to distribute elements such as leaves in a variety of ways. Benefits of this method include the relative ease with which an alpha channel can be created, as well as the level of control the artist has over the exact positioning of the features within the texture.

FIGURE 4. Van Gogh's unhelpful poplars.

Scanned Illustrations

Often overlooked as unsuitable source material, paintings and illustrations of trees can be a very useful starting point for textures. O.K., if you choose Van Gogh's *Poplars on a Hill* (Figure 4) as inspiration, you may find yourself struggling to make use of his mad wavy lines, but there are some examples that are more friendly. Illustrations of different tree species, found in botanical encyclopedias for example, are often highly detailed and accurate. The advantage of this kind of image is that lighting is often very diffuse, without the hard shadows that can make photographs unusable. Also, the foliage is generally on a white background, which makes it easier to separate out than a busy photo.

Photographs

A scanned or digital photo is easily the most common source of textures, and perhaps the most difficult part of the process of turning a photo into a useful foliage texture (see Figure 5) is generating an alpha channel. A solid mass of leaves will be of limited effect when trying to give the impression of complex branching, and so it is important to use transparency within the texture to create the shapes that are far too costly to build into the geometry.

FIGURE 5. Photo to texture (stages one through four, from left to right).

Supposing you have a fairly standard photo of a leafy canopy, the question

then becomes how best to select only the areas that are to be left visible, so that the others can be masked out. I may be wrong, but Photoshop doesn't yet come with a "choose a good selection of leaves" tool. The problem is that most selection methods can't discriminate between the leaves and branches that you want to keep and those which will leave the texture too cluttered.

Unfortunately, as with so many things in life, for a completely satisfying result, you'll have to do the job yourself. Selecting by color and altering the fuzziness, or creating a threshold layer and adjusting the levels, may provide a rough starting point for selection. Still, you'll have to refine these characteristics manually.

Rendered Images

A useful addition to the essentially two-dimensional source material mentioned thus far is the rendered tree. Most major packages have a selection of tree-generating programs with names like "Tree Master Plus" and "Auto Foliage 500." While each is different, they all produce similar results that are too detailed to be used as in-game geometry but can be extremely useful for creating textures. Perhaps the most appealing aspect of this approach is the automatic generation of an alpha channel when the image is rendered, which both saves time and increases accuracy (see Figure 6).

Whichever method you use to create the textures, assembling the geometry on which they will be mapped in order to create the appearance of a tree's canopy must be done carefully and take into consideration the way in which the trees will be encountered. If, for instance, the trees will be viewed from beneath, the orientation of the geometry containing the branch and leaf textures needs to take this into account. The same holds true for the trunk geometry; extra detail can be worthwhile around the lower trunk and root region if players are likely to spend their time in this area.

In the end, finding the best way to make trees is a bit like finding the best way to bake bread. There are a million different recipes. Clever use of texture, coupled with economical modeling, can be the best compromise between detail and practicality, and while the process may be awkward, getting the right result is certainly well worth the effort.

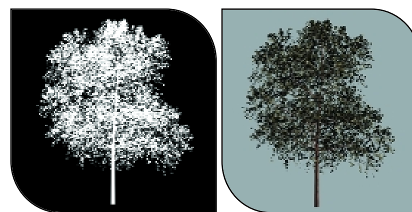


FIGURE 6. Render plus alpha channel equals texture.

brian hixon, daniel martin,

Play by Play: Effective Memory Management

BRIAN HIXON | *Brian has a B.S. in math/computer science from Carnegie Mellon University. After losing his mind from playing too many computer games, Tiburon took pity on him and gave him a job in 1998. Brian is a lead programmer for MADDEN PS2, Gamecube, and Xbox.*

DANIEL MARTIN | *Daniel has a B.S. and an M.S. in computer sciences. After wasting years of his life outside the gaming industry, he saw the light and joined Tiburon in 1998. Today, Daniel is a lead programmer for MADDEN PS2, Gamecube, and Xbox.*

ROB MOORE | *Rob has a B.S. in electrical engineering. After working on graphics chips and APIs for the Nintendo 64 and Gamecube, Rob decided to see why game developers were always griping at hardware guys. He joined Tiburon in 2000 and now it all makes sense; currently he is Tiburon's chief technical officer. Send comments on this article to MMarticle@tiburon.com*

GREG SCHAEFER | *Greg has a B.S. and an M.S. in computer science. Greg spent many years working on telecommunication and network applications prior to joining Tiburon in 1998. He now leads MADDEN PC network development.*

RICHARD WIFALL | *Richard has a B.S. in electrical engineering. He got started in the game industry on 16-bit consoles.*

Back before Al Gore invented the Internet, back when 64KB was more memory than any computer would ever need, there was a time when memory managers didn't exist. But gradually, new computer systems came out with larger amounts of memory (Figure 1). Game designers discovered ways to eat up RAM faster than any system could dish it out. It became obvious that managing all this data with a simple static memory map was too much work. Programmers needed a way to create memory maps that could change dynamically at run time depending on the state of the program. Thus, the memory manager was born.

For an excellent introduction to how a basic memory manager works, check out www.memorymanagement.org/articles/begin.html. This article will assume this basic knowledge and focus instead on things to think about when writing your own memory manager and point out some of the problems you may encounter in using a memory manager. This discussion is based on our experiences in writing and rewriting the memory manager for MADDEN NFL 97 to MADDEN NFL 2002. This article is also slanted toward the C language as written for console game machines. The same principles would apply to C++ and Windows programming, but the details are likely different.

Growing Your Own

So you know that memory managers are great, but what's wrong with just using `malloc` and `free`? They are part of the standard C library, so they should be the best, right? One major problem is that you don't really know what they are doing. If you write your own memory manager, you can generate statistics, add debug code, and add advanced features that might not be found in the default memory manager — features like handles, heap compaction, and virtual memory. The ultimate weapon of game programmers is context. You know the needs of your game and can tailor the memory manager accordingly.

Developing a successful memory manager requires addressing five different issues: ease of use, performance, memory overhead, fragmentation control, and debugging support. Unfortunately, these attributes are all interconnected, and optimizing one can result in substandard results for another. The memory manager designer must govern a delicate balancing act.

rob moore, greg schaefer, and richard wifall

At a lower level, memory manager design also requires paying attention to platform-specific requirements. In addition, it may be possible to utilize hardware support to assist the memory manager.

Ease of Use

With respect to a memory manager, the ease of use consideration really comes down to a single question: Should the memory manager support pointers, handles, or both? When designing a memory manager and dealing with the problem of fragmentation, the use of handles can be very appealing. Unfortunately, while handles provide a straightforward solution to the fragmentation problem, they are much more difficult to use than pointers. A memory manager that supports only handles is essentially pushing its internal complexity back onto the user.

While supporting both handles and pointers is possible, the resulting memory manager is more complicated than one that supports a single method. MADDEN used to support both handles and pointers until an analysis showed that pointers were being used 99 percent of the time. Not surprisingly, when given a choice, programmers used pointers, since they were the easiest solution. Therefore, we simplified the latest MADDEN memory manager by removing handle support and optimizing the pointer support.

Performance

Performance must be addressed both in terms of speed and consistency. A memory manager that is fast most of the time but slows down dramatically at unpredictable times is unacceptable in a gaming environment. Therefore it is important to understand the issues that contribute to performance. From the memory manager user's point of view, two operations will impact the game: allocations and recycling.

Allocation performance is determined by allocation policy, free list management, and the use of garbage collection. The most popular allocation policies are best fit, first fit, and next fit. By organizing the free blocks as a linked list, best fit has consistent O(n) performance, while first fit and next fit have worst-case O(n) and on average O(n/2). By organizing the free blocks as a size-sorted binary tree, best fit has worst-case O(n) and on average O(n log n). By organizing the free blocks as a balanced binary tree, best fit has consistent O(n log n). Garbage collection applies only to memory managers with handle support, and generally involves a fairly significant performance penalty when it occurs, as it essentially makes a copy of the entire memory heap during compaction.

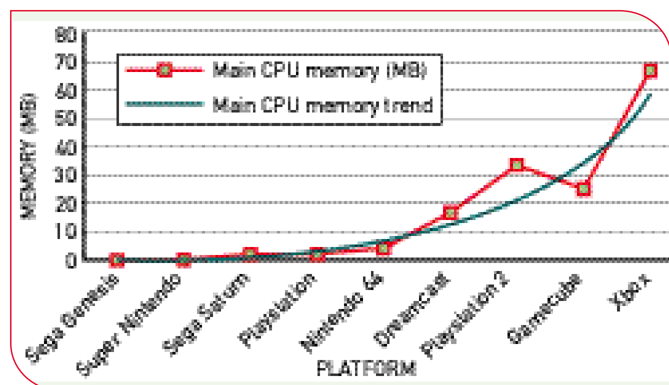


FIGURE 1. Main memory growth for console game machines.

STATIC vs. DYNAMIC MEMORY

STATIC MEMORY
Pros. Fast
Cons. Difficult to maintain
Memory can't be easily reused

DYNAMIC MEMORY
Pros. Objects can be designed to only use memory resources when the object exists
Can support higher level concepts like virtual memory
Easier to support threads
Cons. Fragmentation
Additional overhead

Recycling performance is based on free list management and the use of free block coalescing. Free block coalescing requires the ability to locate the blocks immediately preceding and following any arbitrary block. Some memory structures, such as boundary tag (see Donald Knuth's *Fundamental Algorithms* under References for more information on the boundary tag), allow this in O(1) time, while those that don't require an O(n) scan of the free block list. The current MADDEN memory manager uses boundary tags to allow O(1) access to previous/subsequent blocks and organizes the free list as an unbalanced binary tree. The result is allocation and recycling performance both in the O(n log n) to O(n) range.

Memory Overhead

Memory overhead is the cost we have to pay to manage memory, since each allocation will cost some memory. Memory overhead is an important consideration, especially if the memory manager will be handling a large number of objects. The first decision is whether the memory map state should be stored internally or externally to the memory being managed. If it is stored internally, then the maximum number of allocated blocks does not need to be known in advance, but memory that is not directly CPU-addressable cannot be managed. If it is stored externally, you must know the maximum allocated and free blocks in advance and set aside memory for this state, but address spaces that are not directly CPU-addressable can be managed.

MADDEN previously used external storage for the memory state, but this required additional overhead because it was impossible to predict accurately the maximum number of allocated and free blocks. We had to include a "safety factor," which turned directly into wasted memory. The new memory manager uses internal storage as shown in Figure 2, thus avoiding the entire issue. All allocations are 16-byte aligned and each block has 16-byte overhead. By limiting allocations to a minimum of 16 bytes, every block is guaranteed to have this much storage available. Therefore, when an allocated block is released, those 16 bytes can be used to organize the block into the free list.

It is worthwhile to digress slightly and consider the management of non-CPU-addressable memory. Because consoles are designed for low cost and high performance, they sometimes incorporate special-purpose memory that is not directly CPU-addressable. For example, the Gamecube has 16MB of auxiliary memory (ARAM) that supports only DMA access. A memory manager that stores its state internal to the memory it is managing cannot be used in such cases, while a memory manager that stores its state externally can.

While it may seem appealing to use external state storage in order to support all kinds of memory, our experience with MADDEN has shown this to be a mistake. Memory that is not directly CPU-addressable is normally used only for special types of objects, due to the complexity of using the memory, and often contains only a small number of objects. Therefore, MADDEN now uses a single, general-purpose internal storage memory manager for all CPU-addressable memory and an additional, customized external storage memory manager for any special-purpose case, such as the Gamecube's ARAM.

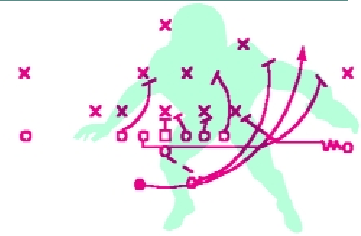
Fragmentation Control

Memory fragmentation is a condition that occurs over time as memory is allocated and released and isolated free blocks form. This is not usually an immediate problem, but as more fragments form and they get smaller, the opportunity for an allocation failure due to fragmentation increases. Eventually, if fragmentation gets severe enough, an allocation may fail because no single free block is large enough to accommodate the request even though the total amount of free memory is (Figure 3). Therefore, a good memory manager needs to either take steps to limit the amount of fragmentation that occurs or be able to consolidate fragmentation periodically through garbage collection.

An allocation failure is often a lethal error for game code and must be avoided at all costs. While it is generally fairly easy to determine the maximum memory required by an application, fragmentation can make such a calculation meaningless. Allocation policy, free block coalescing and multiple heaps all play a part in minimizing fragmentation.

Free memory coalescing is a straightforward technique for limiting fragmentation that attempts to merge a newly released memory block with its neighbors. If the block immediately preceding or following the newly released block is also free, merging the blocks together results in a single, larger free block. This particular technique is almost mandatory, as without it, fragmentation occurs very quickly. In addition, this limits the size of the free list to the minimum number of blocks, which generally has a positive performance impact on allocation. Since this technique has no impact on how the memory manager is used externally, it is incorporated into most memory manager designs.

The choice of allocation algorithm has a definite impact on fragmentation. Unfortunately, because the size and timing of allocations and releases vary for every application and are often different from run to run of an application, it is impossible to say that a particular algorithm is always better or worse than another. However, in general, it has been found



that "best fit" algorithms tend to exhibit the least fragmentation. See Johnstone and Wilson's "The Memory Fragmentation Problem: Solved?" in References for more information on the impact of allocation techniques on fragmentation.

A technique that requires more involvement from the memory manager user is to allocate blocks with similar lifetimes close to each other. If the lifetime of a memory block is known when it is being allocated, the allocation algorithm can attempt to place it by other blocks with a similar lifetime. When these blocks are released, they will be merged into a single large free block. The problem is that this requires the caller to specify both the lifetime of each block as well as the total size of the group of blocks of similar lifetime. As a result, a simpler version of this technique is usually implemented through the use of multiple heaps. By allocating blocks with similar lifetimes within their own heap, a similar effect is achieved, though there are generally practical limitations on the number of heaps that can be effectively utilized.

–12:	header pointer
–8:	type (head)
–4:	data length
0:	header data
+n:	data length
–12:	header pointer
–8:	type (used/free)
–4:	data length
0:	user data
+n:	data length
–12:	header pointer
–8:	type (used/free)
–4:	data length
0:	user data
+n:	data length
...	
–12:	header pointer
–8:	type (used/free)
–4:	data length
0:	user data
+n:	data length
–12:	header pointer
–8:	type (tail)
–4:	data length (0)
0:	data length (0)

FIGURE 2. Memory structure image.

Debugging Support

One of the main benefits of writing a memory manager is the ability to add capabilities that are not provided in the supplied memory manager. By adding some debugging features you can make sure that you are managing the memory rather than stepping in the heap.

To avoid stepping in the heap, you'll need to be able to see the heap. Adding information and tools to help visualize memory usage can be a big help when managing memory. This can be done in a number of ways.

The debugging techniques used by the MADDEN memory manager include the ability to audit memory usage, collect usage statistics, check for consistency, and follow the memory map from a debugger. Auditing is really nothing more than being able to mark a group of blocks when they are allocated and later check to see if they were all released. Usage statistics, such as maximum number of allocated and free blocks, as well as maximum allocated memory, are valuable for evaluating the memory utilization of the application.

Consistency checking goes through the memory map and makes sure there are no detectable errors in it. This means verifying block sizes, checking boundary tags, ensuring matching block counts, and so on. By performing consistency checking prior to every allocation and release (in a debug build), certain forms of memory corruption can be detected quickly.

256 used bytes
32 free bytes
512 used bytes
64 free bytes
96 used byte
80 used bytes
32 free bytes

FIGURE 3. Although 128 free bytes are available, the largest possible allocation is only 64 bytes due to fragmentation.

Also, the memory map contains ASCII debugging information that can be viewed from within the debugger. Prefixing every allocated block with a text string indicating the module and line number of the caller that allocated the memory greatly assists when debugging after a crash.

Naming the memory blocks is a necessity. Imagine going to a store where every item was in a plain box that only had its UPC code and box size printed on it. Finding the item you wanted to buy would be a big challenge. Likewise, if you have a list of memory blocks that only contains their address and size, locating memory blocks is going to be difficult unless you give those memory blocks names. Using the source file name and line number where the allocation occurred would be a good start. In C, this can be accomplished quite easily through the use of macros and the C preprocessor.

Now you can recognize your blocks in a crowd, but it sure would be nice to know who they hang out with. By adding a group label or flags, you can group your allocations at the conceptual level rather than being limited to grouping your blocks by source file. This way you can know that a memory block that was allocated is really being used by your main character, even though the actual memory allocation occurred in your texture library.

Figure 4 shows an example memory dump as seen by a (slightly drunk) debugger. Including debugging information in textual form within the memory map allows you can make sense of it from the debugger. For example, if you had an invalid address access at `0x007cfea4`, you could find that address in the debugger and page around it to see that it occurred in the Joystick Driver library and that the memory in question was allocated by `pJoyDrvr.c` at line 225.

With all this information available, you will need to find ways to view that information that can help you when managing your game. A linear list of memory blocks can be useful for spotting

potential memory fragmentation, while a list of memory blocks sorted by name can be useful when you have memory leaks. If you do find a memory block that has leaked, you will know from its name exactly where the allocation occurred. With group labels you can print out compact memory maps that show how each conceptual module in your code is using resources. By tracking this throughout the project, you can easily spot modules that are using more or less memory than you had budgeted in your original design.

You can also create functions to check whether groups of memory allocations have been freed. This can help prevent memory leaks if you know that in certain situations some groups have no allocations.

Keeping track of worst-case situations is also important. Have the memory manager save the lowest amount of free memory it has ever encountered (update this every time an allocation occurs). If you are using some sort of node system to keep track of your memory blocks, keep track of the highest number of nodes that the memory manager has ever used so that you know if you are close to running out of memory blocks.

Sentinels added to the beginning and end of memory allocations can help detect overflow situations that would normally corrupt other data in memory. If the sentinels don't check correctly when the memory is freed, then some code has been bad.

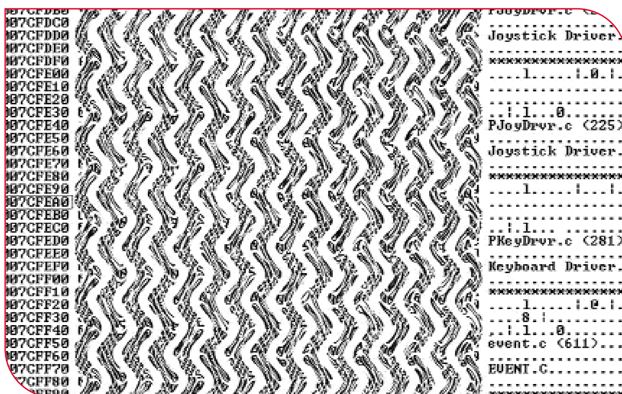


FIGURE 4. Memory blocks with debug information as viewed in the debugger.

Filling memory with recognizable patterns can be extremely useful. We use a different pattern for unallocated, allocated, and freed memory. This way, we can tell in the debugger at a glance what the situation of a particular piece of memory is. When someone forgets to initialize memory they allocated properly, the "allocated" pattern shows up.

You can also have functions that scan free/unallocated memory and make sure that it all still matches the prespecified patterns. If it doesn't match, some code out there is incorrectly writing to memory locations that it doesn't own.

Finally, make sure that you set up an extra heap for debug memory and put all this extra debug information there. You want your game memory to be as similar as possible between a debug and final build.

Platform Specifics

A memory manager presents a logical view of memory in the same way that a file system provides a logical view of a disk drive. Most often, memory managers are concerned with managing dynamic RAM of some sort. Some console makers like to make things more interesting by providing a relatively large main memory but also scattering other smaller chunks of RAM around the system. The memory manager allows us to abstract away the physical details of the memory subsystem and deal instead with a nice, logical model of memory. For example, on the PS2 we don't necessarily need to know that the first megabyte of RAM is reserved for the operating system. It's enough that the memory manager knows. By abstracting away some of the details of the physical memory system, our game can become more platform independent.

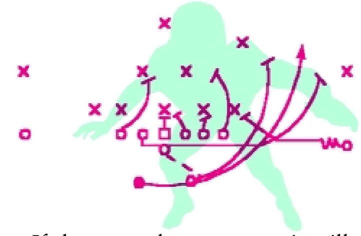
Most console hardware has alignment requirements that, not so surprisingly, differ from platform to platform. Many platforms require anywhere from 4- to 64-byte alignment for buffers in graphics rendering or file IO. Each type of hardware might need the memory manager to be tweaked to better fit the needs and abilities of the platform. Often this information can be passed to the memory manager using a heap setup structure.

Finally, you should be wary of third-party libraries that may use `malloc` and `free`, effectively bypassing your memory manager's interface. The `printf` function in the PS2 standard C library uses `malloc` and `free`; our solution was to write our own `printf` function.

Hardware Support

On most modern computers, the issue of fragmentation has been greatly reduced by the use of a hardware-based paged memory manager unit (PMMU). Obviously, the fact that virtual memory provides an application with lots of addressable memory means that even an inefficient memory manager can be used. However, the more interesting point is that the PMMU without any secondary storage can dramatically help with fragmentation. The PMMU takes a very large address space (larger than the physical RAM it represents) and maps it onto physical memory. Obviously, this mapping is not one-to-one, but rather it maps a subset of the memory space onto a "working set" of memory pages.

The key impact of using a PMMU in terms of fragmentation is that when a memory block is released, any portion of that block completely spanning one or more PMMU pages can be remapped by the PMMU. The result is actually two forms of fragmentation: address-space fragmentation and memory fragmentation. While this effect might seem to make a bad problem worse, it actually simplifies things. Because the PMMU provides a large address space, the address-space fragmentation can be largely ignored. Instead, the allocation algorithm concentrates on minimizing



memory fragmentation by first attempting to utilize the unused portions of allocated memory pages (pages that contain some allocation but are not completely full) before remapping unused memory pages back into memory.

Managing Your Memory Manager

Memory managers are like most real-life managers. You have to keep them under control, or else they tend to wander off without adequate information and make strange decisions. Your memory manager sometimes needs your help to stay on the right track too. Let's look at some common problems that can occur and how to work around them.

Fragged again. One of the major side effects of hand grenades and memory managers is fragmentation. Previously, we discussed fragmentation in the context of designing the memory manager to avoid or reduce the effects of fragmentation. However, there are also some application-level techniques that can reduce fragmentation.

The use of static allocation (memory defined within a module with storage allocated within the application image) avoids the memory manager completely and thus avoids fragmentation. Of course, this usage really only makes sense when a single object will be represented and when the lifetime of that object is approximately the duration of the entire application. In such cases, static allocation can provide a benefit by limiting utilization of the dynamic memory manager to those memory objects that are truly dynamic.

Another strategy that relies entirely on the memory manager user is to perform submanagement of memory based on specific program knowledge. For example, if a module knows that it needs to allocate X objects of size Y, it may be far more efficient for the user to allocate a single block of X * Y bytes and perform its own management within that larger block. By reducing the number of blocks that the memory manager has to deal with, the user has generally made the job of the memory manager easier. There is, of course, a caveat. Depending on the amount of fragmentation and the size of X * Y, it is possible that the application could find itself in the situation where an allocation of X * Y fails due to fragmentation, whereas X allocations of Y would have succeeded. We also try to discourage this practice when possible, as there is code-maintenance overhead.

One way to help avoid memory fragmentation is to always free memory in the opposite order from which it was allocated. In fact, if you were always able to allocate and release in such an order, you would never have memory fragmentation. Realistically, it's not possible to follow this practice all the time, but doing it as much as possible will help.

Memory fragmentation is going to occur, and at some point you will probably run into a situation where it causes a problem in your game. You might have fragmentation that is occurring in one part of your game that is causing a memory allocation to fail in a totally unrelated area. Fragmentation might even manifest itself as a situation where your game will fail only when arriving at part of your game through a specific path (that causes the fragmentation). If you don't have some advanced form of garbage collection, you are going to have to use other, more crude methods to limit this problem.

One possibility is to change the code that is causing the fragmentation to use a different allocator so that it doesn't cause fragmentation. A common way to accomplish this is to have an allocator that allocates from the other end of memory. Depending on your game, fragmentation can become more problematic over time (especially if your allocations don't have close locality of lifespan). You can use brute force to minimize these effects, such as shutting down and restarting code modules between levels as a brute-force garbage collection technique.

Release builds. When running a release build, there isn't any debugging information in the game, as it will consume extra memory. But you still need a way to know where the game runs out of memory. For MADDEN, we assume in the code that we will never run out of memory. If the game does run out, it will crash and optionally dump some memory info. With a special build, we display some partial information about the memory manager and the game state so that we can determine if it ran out of memory because of fragmentation or other reasons.

Getting on the Field

When talking about memory management, programmers often resort to words more often associated with cow pastures than games. Terms like heaps, stacks, and chunks are thrown around like flies buzzing around you-know-what. To see how important a memory manager is to a game, you have to get past the abstract poo. A good memory manager allows you to have more animations, more characters, more textures, more sounds — in short, more of everything that your game-playing customers love.

In this article we have described some of the issues that may come up in writing and using your own memory manager. After years of writing and rewriting memory managers for the MADDEN series, one piece of advice that bears highlighting is simply to make sure you schedule adequate design time on this very important piece of your system; you'll be glad you did.

REFERENCES

The Memory Management Reference Beginner's Guide: www.memorymanagement.org/articles/begin.html

Johnstone, Mark S., and Paul R. Wilson. "The Memory Fragmentation Problem: Solved?" In *Proceedings of the International Symposium on Memory Management.* ACM Press, 1998. pp. 26–36. www.cs.utexas.edu/users/wilson/papers/fragsolved.pdf

Knuth, Donald E. *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms.* Reading, Mass.: Addison-Wesley, 1973. pp. 435–444.

david freeman

Four Ways to Use Symbols to Add Emotional Depth to Games

Max Payne. Elite Force. Thief. Ico. Deux Ex. Oddworld. Medal of Honor. Baldur's Gate. The more recent Final Fantasy games. More and more developers are pushing the game design envelope, forging new entertainment experiences and art forms that draw on the roots of traditional gaming, but also partake of more sophisticated storytelling and characterization.

As the production values in games continue to soar, the trend toward equivalent advancement in storytelling is inevitable. For game designers involved in creating each successive advancement, these are exciting times.

Remember in *Braveheart* when Mel Gibson charged into battle holding a handkerchief his wife gave him before she was murdered? That handkerchief is a symbol. This article will explore four different ways to use symbols to evoke emotional response from an audience.

But first, let's look at some of the fundamental issues relating to the role of emotion in games.

Why Put Emotion into Game Stories?

This is an important discussion, and probably one that deserves its own article. But, in a nutshell, other than the inherent joys of creating a rich work of art, the reasons also boil down to potential profits.

First of all, many more people watch film and television than play games. Most will never be lured into playing games until games begin to offer the emotional range and depth of the entertainment that they're used to enjoying. Also, a more involving game experience means better word of mouth and more buzz. The press likes to write about these kinds of games, which results in more sales. Seeking out better profits also means staying ahead of the competition. Certain game developers are working hard to advance emotion in gaming. Those creating games with stories and characters without investing in putting emotional depth into their games will find themselves further and further behind, and their games will be eclipsed. And, the better game visuals get and the more games look like films, the more people will want to compare them to films. Thus, weak writing and shallow emotional experiences in games featuring stories and characters will increasingly stand out negatively in consumers' minds.

Many of the challenges that designers face in creating emotionally rich game experiences have already been addressed in other media. Traditional screenwriters, deprived of the game designer's ability to actually insert an audience into a film, have figured out perhaps thousands of ways to induce emotional involvement. Game designers will want to test the applicability of these techniques to their new games and modify them so they'll work within an interactive experience.

A big part of successful communication between a writer and his or her audience is writing outside of the audience's conscious awareness. No one expects the game player to pick out every sound used in a game's sound design, nor every instrument utilized in a piece of music, nor every tiny shadow. So too, an extraordinary amount of what a writer

DAVID FREEMAN | *David contributed to the script for The Matrix sequel game (in production at Shiny Entertainment). As a writer and producer, David has had scripts and ideas bought or optioned by MGM, Paramount, Columbia Pictures, Castle Rock, and many other film and television companies. David teaches "Beyond Structure" (www.beyondstructure.com), a popular Los Angeles–based screenwriting class, which has been taken by writers from many top films and television shows, as well as by many well-known game designers. He welcomes your thoughts on this article at freeman@dfreeman.com.*

Illustration by Ben Fishman

does is designed to affect a game player emotionally but not be consciously noticed. This article will focus on the use of symbols, which are almost always employed in a way so that they're just on the edge, or preferably just outside, of a game player's conscious awareness. A workable rule of thumb is that no more than 25 percent of the players who come upon a symbol should be consciously aware that it actually is a symbol.

The five arenas of "deepening." I use the phrase "deepening techniques" to describe all those writing techniques that impart a sense of depth to a piece of dialogue, a character, a relationship between two or more characters, a scene, or a plot. Other words that mean something similar to deepening include poignancy, soulfulness, layers, and emotional or psychological complexity. When people talk about these things, they're talking about what I call emotional deepening. Symbols are always a deepening tool.

One game designer who has taken some of my story and writing workshops pointed out that to focus on more subtle or sophisticated techniques such as the use of symbols is putting the cart before the horse. Many game designers might benefit from learning more basic techniques for creating rich, complex, and compelling characters and natural dialogue. This is true. But one nice thing about symbols is that, with very little effort, you can easily and precisely enhance the depth of your scenes and plots.

When you create a symbol, you're not trying to create an intellectual puzzle in which the player tries to figure out what the symbol means. Such an intellectual exercise would work directly against the goal of increasing emotional immersion. Instead, symbols, when employed artfully, should evoke emotions — even though, when you do your work well, most players won't consciously notice the symbols that you use. It's not necessary for a game player to notice a symbol in order to be emotionally affected by it.

It's certainly O.K. that a small percentage of players who consciously notice your symbol might stop and think about the symbol's meaning or meanings. But it's only acceptable if, at the same time, the symbol generates in those players an emotional experience as well. Following the guidelines in this article will help ensure that this is what the player actually experiences.

Another advantage to using symbols in game design is that games often offer an opportunity that films do not. In film, symbols, when used artfully, enhance emotional depth. As we'll see, when used in games, symbols can not only perform this function, but can also be used or given a function in gameplay as well.

Symbol Type #1: Symbol of a Character's Condition or Change in Condition

This use of symbols is what I call a scene-deepening technique, because you use it in a specific scene and might never use the same symbol again. Its use can be either visual or verbal, meaning that there must be either something visual on screen or something said by one of the characters that reflects what an on-screen character is going through emotionally.

Example #1: Visual. In a particular episode of *Star Trek: Voyager*, Captain Janeway (Kate Mulgrew) finds herself in an extended battle with the captain of a rogue Federation ship. The captain and crew of that ship are killing harmless aliens in order to use the chemicals in the aliens' bodies to propel their ship. But Janeway herself becomes so obsessed with stopping the rogue captain at any cost that she crosses the bounds of ethics and good judgment and imperils her crew. This conflict generates a series of arguments with Chakotay (Robert Beltran), her first officer.

A metal plaque with the words "U.S.S. Voyager" falls off of Voyager's bulkhead during the battle with the rogue ship. This plaque is a symbol that

the spiritual core of Voyager — including the moral codes of the Federation, the Starfleet tradition of honor and humanity, and the moral center of the people who uphold these codes and traditions — has been damaged. It's a symbol of Janeway's and Chakotay's conditions or changes in condition.

The plaque falling off of the bulkhead affects us emotionally. If viewers make only an intellectual connection between the plaque and the abandoned Federation values, then the writer hasn't been artful enough in his or her creation of the symbol.

Example #2: Visual. The 1957 war film *Bridge on the River Kwai* won many Academy Awards and still stands up as a masterpiece. Alec Guinness plays Colonel Nicholson, who commands a group of British soldiers captured by the Japanese and forced to work as slaves in a POW camp in Burma. I won't reiterate the convoluted plot, but in short, due to his ego, Nicholson has his men help the Japanese build a strong and beautiful bridge. In effect, he has helped the enemy. But, near the end of the film, during a battle at the bridge, he has a powerful revelation, and says, "What have I done?"

At that exact moment, he reaches up and touches his commander's cap. This is a symbol of the character's condition or change of condition. His touching the cap is a symbol of his changing back to becoming what he once was — an honorable British soldier.

An explosion goes off nearby that knocks him to the ground, wounded by shrapnel. When he stands up, his cap lies on the ground, but he's too dazed to notice immediately. He reaches for the top of his head and realizes that the cap is gone. He then bends down and picks it up off the ground. His reaching toward his head for the cap, and then his picking it up off the ground, again is the same kind of symbol, signifying that he's become the honorable man he once was.

He puts his conversion into immediate action. As he dies from the shrapnel wound, he directs his fall onto a dynamite detonator, which in turn blows up the bridge he had so painstakingly built.

In Sony's ICO, the main character's quest to save a beautiful girl with mystical powers contains symbols that engage the player's emotions and affect gameplay.

As was the case with the *Voyager* example, most people in the audience wouldn't consciously notice this element. And yet it would still contribute to the depth of the audience's emotional experience. It's a strange moment for a writer when he or she realizes that a great deal of writing involves trying to create emotional effects that no one will consciously perceive, perhaps ever.

Example #3: Verbal. Perhaps you saw the provocative film *American Beauty*, in which Wes Bentley plays Ricky Fitts, a teen without fear of social pressures, who has an honest appreciation for the beauty all around him. He seems, in some ways, to be enlightened.

Contradicting his supposed enlightenment is the fact that he sells drugs, is completely emotionally detached, and is fascinated by death. In fact, his veneer of serenity is what I call a "mask," or a false front. (Masks, in all their various forms, are very sophisticated character-deepening techniques.)

At a certain point in the film, Lester Burnham (Kevin Spacey) drops in on Ricky to buy some dope — in particular the really potent stuff that he'd smoked with Ricky a few nights earlier. Ricky pulls out a bag of dope and explains that it's ". . . top of the line. It's called G-13. Genetically engineered by the U.S. government. Extremely potent. But a completely mellow high, no paranoia."

LESTER: "Is this what we smoked last night?"

RICKY: "This is all I ever smoke."

Why is this a verbal symbol of a character's condition or change of condition? Because Ricky, unknowingly, has just described himself. Ricky had been a passionate young man, until his father, as punishment, had him committed to a mental institution for two years, where he was heavily drugged. This experience broke his spirit. So Ricky himself has been government-engineered, and his fake serenity (his mask) is that of a "completely mellow high." But like all chemical highs, the effects aren't real.

Example #4: Verbal. Sometimes, in the television business, you need to write a sample script just to show that you can adapt your writing style to different shows. I recently wrote a sample *X-Files* script. In the story, Mulder no longer fits in professionally with Scully and

Doggett. He had always been driven in his paranormal quests by the search for the truth about his missing sister. But, with that case solved last season, he no longer has a dream or ambition to push him forward.

In the middle of a conversation with Scully, Doggett, and Skinner, Mulder notices Skinner's office clock. Checking it against his own watch, he says, "Is that clock right?"

No one responds to the question — the conversation merely proceeds. (Quite frequently, in dialogue, not every statement or question gets a response.) Why the throwaway line about the clock? It's a symbol of Mulder's condition or change in condition. In this case, it symbolizes that he's out of sync, or out of step, with all the others. In effect, his time has passed.

Will anyone reading the script consciously note that line of dialogue? Unlikely, any more than they would note Wes Bentley's line in *American Beauty* about the government-engineered pot. As with the other examples, the symbol operates outside of the audience's conscious awareness.

Game example. In the game ICO, a boy in a faraway land helps lead a beautiful girl with mystical powers out of a towering castle where both are trapped. He bravely overcomes many terrifying obstacles in his journey, which is more focused on freeing the girl than himself.

Near the very end, he gets a magical sword that crackles with a kind of spiritual electricity. This is a symbol of the boy's condition or change in condition. It symbolizes that he's attained a level of power; the demonic creatures that once attacked him now flee him and the sword. And it symbolizes that he now belongs with the girl, for the electricity that the sword exudes looks exactly like the mystical energy that the girl can wield when she needs to, and which has the same magical abilities.

Since the boy uses the sword to accomplish his final tasks, this is what I call a usable symbol. It serves double duty by both working to deepen the emotional experience and also playing a role in gameplay.

Hypothetical game example #1. Let's say we have a sword-and-sorcery game in which, during a fight to save some villagers, the wisest and most beloved village elder is killed. The villagers are stunned. A cloud could pass in front of the sun at that point, throwing a shadow over the village (during either a cinematic sequence or gameplay). The shadow would symbolize the villagers' sadness — and perhaps yours as well, if you had found the old man endearing (and you would have, if the character was rich enough and the dialogue was compelling).

Hypothetical game example #2. After great effort and many struggles and battles, you have attained the highest rank a warrior can attain. At that moment, an eagle flies diagonally overhead in the sky. It's a symbol of your lofty achievement.

It's important to reiterate here that it doesn't matter if no one consciously notices the impact of these symbols. They deepen the experience nonetheless.

Symbol Type #2: Symbolic Subplot

Usually at least one of the characters (although sometimes more) in a story has what I call an emotional fear, limitation, block, or wound. Quite often, this person is the lead character, although not necessarily.

In the first *Star Wars* movie, Luke Skywalker had to learn who he was (a Jedi knight), Han Solo had to learn responsibility and how to act as a member of a group (instead of operating solo), Princess Leia had to learn to be vulnerable in love, Obi-Wan had to learn he could still make a difference, and C-3PO had to learn courage. Each of these characters was forced to confront their respective fears, limitations, blocks, and wounds (FLBWs, for short).

Usually, the character doesn't know he or she has an FLBW. If you pointed it out, the character would probably disagree; in fact, they're usually quite oblivious. It's unlikely, for instance, that Han would have agreed with you if, at the start of the film, you accused him of being unable to function as part of a team. It's unlikely Luke would have agreed if, at the start of the film, you accused him of having no idea who he was.

A character's path of growth through his or her FLBW is a rocky one; quite often the character resists growing. A character's path of growth through the FLBW is called a character arc. In many stories, some of the most compelling emotional moments are wrapped around a character's process of wrestling with and eventually growing through his or her emotional fear, limitation, block, or wound.

Some writers insert a symbol into the story that represents the character's arc. That is, as the character changes and grows, the symbol changes right along with the character. Therefore, a symbolic subplot is a plot-deepening technique because it continues throughout all or most of the plot (unlike the symbol of the character's condition or change in condition, which occurs in a single scene or a small

part of the plot).

Example #1. In the new *Star Trek* series, *Enterprise*, one of the crew, Ensign Hoshi Sato (Linda Park) is a woman with extraordinary linguistic abilities. In one of the early episodes, she's having a hard time adapting to life on a starship. She wants to go home, back to Earth.

She has brought a pet along with her — a yellow slug. The slug isn't doing well aboard the ship. Environmental conditions threaten its health.

By the end of the episode, after discovering how much the crew needs her, she has made her peace with being in space. She drops the slug off on an Earth-like planet, where it will survive just fine.

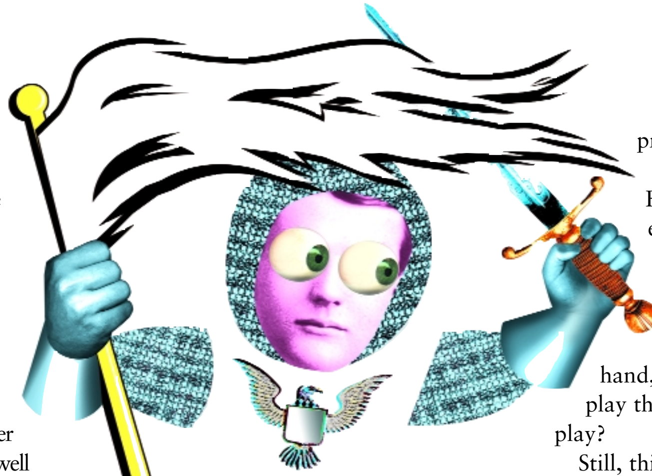
Thus the slug is a symbolic subplot. The slug not doing well in space equates with Sato not doing well in space. The slug being put on a new planet and doing well there thus equates with Sato surviving and thriving away from Earth.

With a symbolic subplot, the audience can stay abreast of a character's progress in his or her character arc just by checking up on what's happening with the symbol. Just as in the case with the symbol of a character's condition or change in condition, a symbolic subplot may or may not be consciously noticed by the audience or game player.

Let's revisit the example from the *Enterprise* episode. In this case, unlike most, we are quite aware that the slug is a symbolic subplot, for the doctor on board the ship even points this out to Ensign Sato. While speaking to her, he compares her difficulties to those experienced by the slug.

This bit of dialogue violates the guideline of having the symbolic subplot operate just outside of most people's conscious awareness. In my opinion, this was a mistake. The slug symbol would have generated more emotion if it hadn't been pointed out to the audience. "Look, here's a symbol" is usually not the best way to go. However, as every writer knows, to every guideline there are always successful exceptions.

Example #2. In the film *Wonder Boys*, Michael Douglas plays a character who wrote a great novel decades ago and is now a washed-up creative writing professor at a prestigious liberal arts college. His life's a mess. He's depressed, and he's been working forever on a sprawling novel that he hasn't shown to anybody.



The symbolic subplot is the novel he's writing. The novel is analogous to his life. We learn that the he's been working on the book for decades. Then we learn that it's a sprawling jumble, with plotlines going off in all directions but no focus, just like his life. It comprises tons of details without a unifying thread, just like his life.

Further along in the film, the pages of his manuscript — the only copy he has — are blown to the wind (symbolic of his life falling apart). Later still, when someone asks him what the novel was about, he can't answer — meaning he has no idea what his life is about. By the end, once he feels his life has again assumed meaning and direction, he starts a new novel, a novel that has power and focus.

Using this technique in games. Trying to build in a character arc for your player opens up a can of worms, because in a symbolic subplot, the changes in the symbol reflect the changes that your character undergoes as he or she progresses through the rocky path of his or her character arc. And how do you manage how a character goes through a character arc when that character is controlled by the game player?

This question takes us right to the cutting edge of story-based games. To explore all the ways in which game designers are tackling or could tackle this problem would be an article in itself, if not several.

Furthermore, it opens up another problem. On one hand, how do you tempt players into seeing themselves in a role and making decisions appropriate to that role? On the other hand, how do you allow players to play the game the way they want to play?

Still, this is one direction in which story-based games are moving. For instance, let's take Raven Software's action-adventure game STAR TREK: VOYAGER — ELITE FORCE. The game tries to create a character arc for Alex, the main player character (what I call a "first-person character arc," since the person who's supposed to undergo emotional change is the player). The attempt to cast the player as the Alex character, thereby helping the player to experience character growth during the story, is done through a variety of methods: observation of a character's behavior and speech during cinematics; watching how other characters respond to the player character; hearing the words coming out of the player character's mouth (what I call "self auto-talk"), spoken in Alex's voice and with his personality; and the player's changing responsibilities as the game progresses.

While these first-person character arcs are a fascinating and critical area of discussion, I'll bring the subject back to where we began. How can a designer use a symbolic subplot to deepen a plot by echoing a player's first-person character arc?

Let's imagine a game in which the player is a samurai swordsman. He's a master of many weapons. Armed with a full range of finely honed steel instruments of death, he leaves his samurai master's training to rescue his master's niece from an evil warlord. This mission will set a much bigger plot in motion.

The obvious character arc follows the player character from his origin as a novice swordsman to becoming a master himself. Because this is the most typical character arc, let's toss it out. As I often

tell my writing students, when it comes to characters, lines of dialogue, scenes, or plots, a good general guideline is, "Find the cliché, then throw it away." (This guideline also dictates that the master not be a clichéd wise Asian character either.)

So let's make our character's arc to "attain a spiritual connection to the universe." As the samurai character attains spiritual wisdom or abilities, perhaps the world will start looking different in some way. Perhaps he'll be able to perform extraordinary moves akin to those demonstrated by the fighters in *Crouching Tiger, Hidden Dragon*. Could you give this character arc a symbolic subplot?

Hypothetical game example #1. Perhaps the samurai's master has given him a sword. It makes a harsh, ringing noise when swung. But as the player character progresses along the character arc, the noise becomes beautiful and harmonic.

Hypothetical game example #2. Suppose the player character recharges his life force by returning to a beautiful little bamboo meditation hut suspended over a small stream. In the beginning of the game, the stream is muddy. But as the player progresses along the character arc, the stream gets clearer and clearer.

In either of these two examples, the player may or may not notice the change in the symbol. This situation is just what a game designer generally wants: a symbolic subplot that works just at the edge of the player's conscious awareness or just outside of it.

Making usable symbols in gameplay. In the first example, perhaps when the sword makes its most beautiful, harmonic sound, something extraordinary happens. A frail old man in the village is, in fact, much more than the peasant he appears to be. When he hears that beautiful sound, he knows the samurai is spiritually ready and gives the player character some special weapon, amulet, potion, or secret that is essential to the accomplishment of the game's final and most dangerous task. Or, taking a cue from ICO, perhaps it's only when the sword makes this beautiful sound that it's fully charged and thus useful against the final and most formidable enemy.

You could also find a way to turn the river (in the second example) into a usable symbol. Maybe the master built the meditation hut over the river and imbued it with magic of which the player character is unaware. Let's say the master dies during the course of the game. But, when the character arc is complete and the stream becomes clear, the master's face can be seen in the river, from which he dispenses advice that is crucial to accomplishing the game's final tasks.

A symbol doesn't need to be used in gameplay to justify its being there, for its main purpose is to enhance the depth of the emotional experience. However, a symbol that can also function as an element of gameplay obviously represents an opportune situation.

Game example. In the game AIDYN CHRONICLES: THE FIRST MAGE, one of the player character's close friends is an NPC who's a reluctant knight. Though the knight has sworn off the violence of battle, he's continually forced to fight for his king, for honor, and to support an honorable cause. He carries a pole bearing the banner of the kingdom he serves. As a tool of gameplay, the banner has certain protective functions. Because of this, the banner is often ripped in battle, symbolizing that the knight's heart is torn every time he violates his decision to abstain from fighting. Furthermore, the banner, when torn, prompts discussions by the knight and those around him as to the ethics of his fighting in battle versus being a man of peace. The banner is a symbolic subplot, indicating, at any given moment, the knight's state of mind as he wrestles with the decision to be, or not to be, a warrior.

This is one of those examples in which a symbol serves a double duty. Not only does it deepen the emotional experience, but it also is a usable symbol with a function in gameplay.

In the first-person action game STAR TREK: VOYAGER — ELITE FORCE, players assume the role of a character named Alex. One way we know Alex is growing in maturity and wisdom is that he's given more and more responsibility for the Elite Force team.

Symbol Type #3: Foreshadowing

Foreshadowing is another plot-deepening technique. Although it only appears in one specific scene, it prepares

This billboard is littered across the rooftops in MAX PAYNE. As the story evolves, so does the significance of the associations the slogan carries, heightening the player's emotional involvement in the game.

us for a later plot development. In foreshadowing, once again you're creating a symbol that usually operates outside the conscious awareness of the player or audience. The symbol, or what occurs to the symbol, suggests something that will occur later in the story to one of the main characters — usually something bad.

Example. In the film *The Shawshank Redemption*, Tim Robbins plays a man who has been unjustly sent to prison. There he runs afoul of the warden, and the two become enemies. Later in the film, another man who has information that could clear Robbins is sent to the prison. The warden finds out about this and asks the man to step out with him into the prison yard at night. The warden grills the new prisoner, who confirms his knowledge of information that could help Robbins.

The warden, finished with his inquiry, tosses his cigarette on the ground and steps on it to put it out. He walks away, and the prisoner is shot from an unseen source in a guard tower. The extinguishing of the cigarette was the foreshadowing that the prisoner, or at least the information he had, was going to be snuffed out. As such, it evokes an ominous feeling when we see it happen.

Hypothetical game example. Let's go back to our samurai swordsman. His master has a bonsai tree that is 150 years old, cultivated and handed down to him by his own master, who is long since deceased. The samurai's master has used the careful cultivation of the small tree to perfect his patience.

Then, either during a cinematic or during gameplay, the villain destroys the tree. This would foreshadow the master's impending demise.

The bonsai tree could also be turned into a usable symbol with a function in gameplay if its magic heals the samurai when he's injured or restores his life force when it's been depleted. Thus, the tree's destruction would not only foreshadow the master's death, it would also affect gameplay by depriving the samurai of a source of healing and thus increasing his jeopardy.

Symbol Type #4: A Symbol That Takes on More and More Emotional Associations

This is another plot-deepening technique, as it too tends to extend throughout an entire plot. It can be either a visual object or a verbal phrase.

One symbol of this type is a very familiar one: the American flag. What does the flag mean? It means a lot of things: democracy; courage; the right to live the life you choose; freedom of speech and religion; a nation ruled by law; Yankee ingenuity; and more. Yet when we look at the flag, we don't consciously think of all these things, we just experience the emotions that these associations evoke in us.

When a symbol reappears over and over again during emotionally charged moments, some of the emotion rubs off on the symbol, and the symbol thus takes on more and more emotional associations as the plot advances.

Visual example. In the film *Braveheart*, Mel Gibson plays William Wallace, a historical revolutionary leader in Scotland. There's an interesting symbol used throughout the film — a thistle, and a handkerchief with a picture of a thistle sewn into it. This symbol takes on more and more emotional associations as the film goes along.

When Wallace is young, a little girl, Murron, gives him a thistle at the funeral of his father and brother, who have been killed by the English. So the thistle is associated with love. When they're older, the two begin dating, and he gives her back this same, dried thistle. Once again it is associated with love. When Murron marries him, she gives him a handkerchief with a picture of a thistle embroidered on it. It is still associated with love.

Later, Murron is murdered. Had this been the only way the handkerchief had been used, whenever Wallace looks at it with sadness, we would understand and feel his personal anguish. It would evoke in him (and in us) emotional memories

and feelings about her uniqueness, the beauty of their love, and the sadness of her passing.

At this point, we could call this a highly personal symbol, as it would be highly personal to him for reasons we can understand and which move us too. A highly personal symbol, and a character's reaction to it, can be an effective way to evoke a lot of emotion. It's a character-deepening technique. However, in *Braveheart*, the handkerchief goes on to take on more and more emotional associations throughout the plot, and so it becomes a plot-deepening technique.

After killing the English magistrate who had murdered Murron, Wallace stares at the handkerchief. By now it's begun to be associated with revenge. The handkerchief will be with him as he becomes a leader of the Scots in their fight for independence, so it eventually comes to be associated with freedom. And finally, after Wallace is killed, wishy-washy landowner Robert the Bruce takes up the fight. Robert leads his men into battle holding the handkerchief, which is now associated with courage.

Throughout the film, the handkerchief with the thistle keeps reappearing, always during emotionally charged moments and always associated with love, revenge, freedom, or courage. By the end, the handkerchief is simply saturated with emotional associations, sort of like the American flag. An important point to make here is that when we see the handkerchief in *Braveheart*, we don't consciously think about all of these meanings and associations. Instead, the handkerchief evokes feelings in us from the many emotional experiences with which it has come to be associated.

Hypothetical game example: Visual. Let's say you're designing a game with a Tolkien-like story. (Yes, it's overdone, but we're just talking hypothetically.) So you've got your meek, Hobbit-type player character going up against a fearsome enemy with supernatural powers. Maybe the player character's motivation is that the villain wiped out his family. His father had given him a pendant with his family crest, handed down through the generations.

The first time we see the pendant is in a cinematic, when the father, as he lies dying, gives it to the son. So the pendant is associated with love. As the player character goes on his quest to bring down the villain, he can recharge his life force (if he doesn't do it too much) by clenching the pendant. So the pendant comes also to be associated with life. At some point the player character needs to give the pendant to a fallen, dying friend, to save her by recharging her life force. Now the pendant is associated with the act of self-sacrifice for a friend. And if the pendant eventually comes back to the player character and gives him a decisive superboost of life force for the final battle, it would then be associated with victory.

Although it would operate outside the player's conscious awareness, the pendant would be a symbol that takes on more and more emotional associations, thereby adding emotional depth to the story. However, because the pendant also plays a role in gameplay, it's doing double duty as a usable symbol.

Game example: Visual and verbal. In MAX PAYNE, above the rough-and-tumble squalor of the city float billboards for the mysterious Aesir Corporation, with its logo (the *r* in Aesir has a little wing on it) and its slogan, "A bit closer to heaven."

At first, the billboards have the emotional quality of taunting the residents of the city by reminding them of class distinctions. After Max (the main character) discovers that the Aesir Corporation is responsible both for the city's decrepit condition and the murder of his wife and child, the logo and slogan become associated with the enemy. And when Max triumphs in the end and finally attains some inner peace, he adopts the slogan "A bit closer to heaven" as his own. The phrase is now associated with transcendence.

If this symbol only made MAX PAYNE players think about these different associations, then despite the fact that it was a wonderfully bold and inventive attempt, it was, to a great degree, unsuccessful. But if it evoked in players a variety of emotions that accompanied these different associations, then it was successful.

Going Deep

This article has covered four distinct techniques for evoking emotional depth with symbols. Each use is quite different from the other, and they can be used in combination. If no one notices your work after it's done, that's just fine — in general, they're not supposed to notice.

When using symbols, you're not creating intellectual exercises for your audience, forcing players to try to figure out what a symbol means. Using a symbol for that kind of mind game would detract from any emotional impact. Instead, when you use one or more of the techniques presented here, you deepen the player's emotional experience in the game by letting the symbol evoke the player's emotions.

While many of the examples of these techniques come from film, their use in games presents a unique tool to designers in the form of usable symbols functioning in gameplay. Games with stories have come far, but still have a distance to go. When game designers and writers master techniques to create complex characters and artfully evoke emotions during

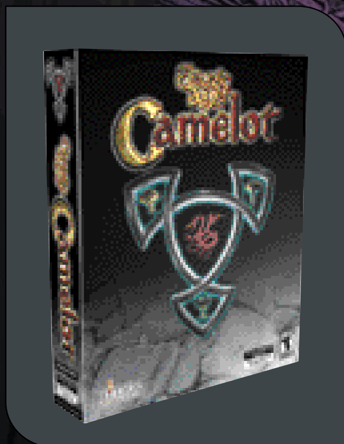
ACKNOWLEDGEMENTS

Thanks to Wagner James Au, David Perry, Chris Klug, Jason Bell, Henry Jenkins, Mike Morhaime, and David Taylor for their very helpful feedback in preparing this article.

play, this new entertaining art form will truly have come into its own.

matt firor

Mythic Entertainment's DARK AGE OF CAMELOT



GAME DATA

PUBLISHER: Mythic Entertainment/Abandon Entertainment/Vivendi Universal Interactive Publishing

NUMBER OF FULL-TIME DEVELOPERS: 25

NUMBER OF CONTRACTORS: 5

ESTIMATED BUDGET: $2.5 million

LENGTH OF DEVELOPMENT: 18 months

RELEASE DATE: October 9, 2001

PLATFORMS: Windows 98/ME/2000/XP

DEVELOPMENT HARDWARE (AVERAGE): 900MHz Pentium IIIs

DEVELOPMENT SOFTWARE: 3DS Max, Photoshop, Visual C++, Linux GNU C++, various proprietary in-house tools

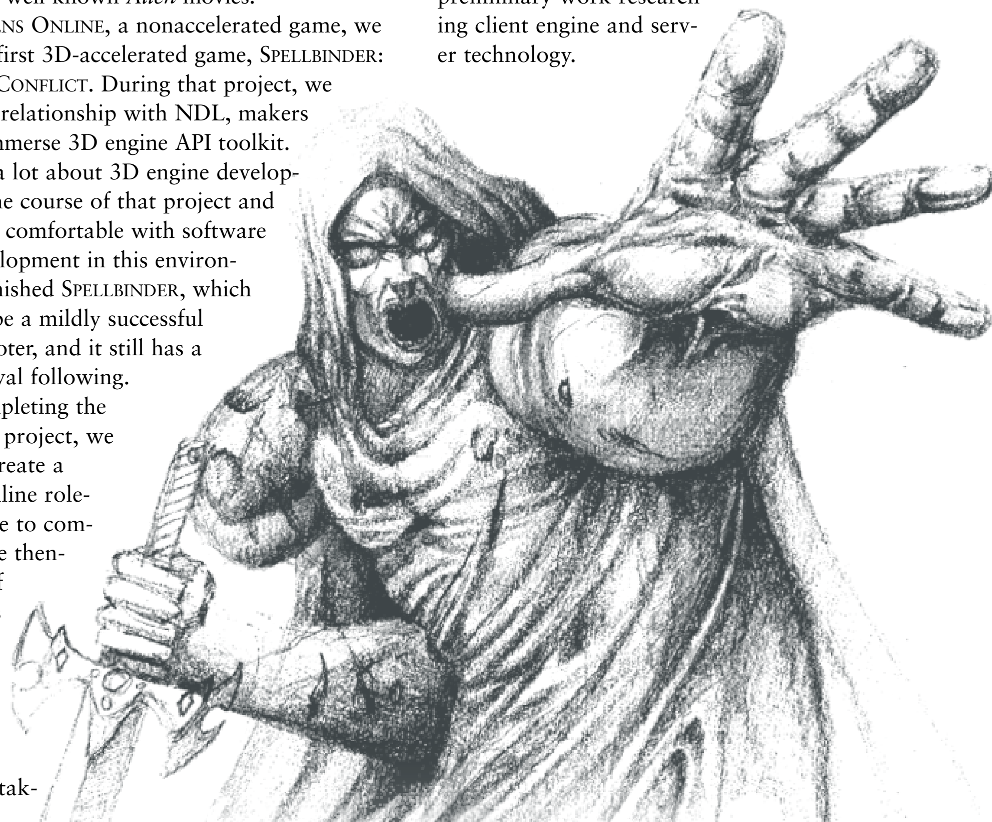
NOTABLE TECHNOLOGIES: NetImmerse, Linux open-source server and database products

DARK AGE OF CAMELOT was the best-selling computer game in the United States for the week of October 7, 2001, and is still comfortably in the top five as I write. This Postmortem is an overview of how this successful title was conceived and developed. My role on the project was as the game's producer.

Mythic Entertainment has been developing online games as a company since 1995 — forever in this field — but the company's founders had made online games even before then. In fact, as a company, we probably have more experience than any other company in developing online games of all types — over the years we have developed role-playing games, first-person shooters, top-down spaceship shooters, and strategy games. When I last wrote a Postmortem here in the pages of *Game Developer*, it was back in May 1998 for ALIENS ONLINE, our online first-person shooter based on the well-known *Alien* movies.

After ALIENS ONLINE, a nonaccelerated game, we created our first 3D-accelerated game, SPELLBINDER: THE NEXUS CONFLICT. During that project, we developed a relationship with NDL, makers of the NetImmerse 3D engine API toolkit. We learned a lot about 3D engine development over the course of that project and became very comfortable with software and art development in this environment. We finished SPELLBINDER, which went on to be a mildly successful Internet shooter, and it still has a small but loyal following.

After completing the SPELLBINDER project, we decided to create a graphical online role-playing game to compete with the then-new wave of online RPGs such as ULTIMA ONLINE and EVERQUEST, which were taking traditional text-based games and adding a graphical front end, with very successful results. Over the years, we had developed several nongraphical online role-playing games, including DRAGON'S GATE and DARKNESS FALLS: THE CRUSADE. Because of our experience developing RPGs, we knew that we had to have a slightly different slant on our new title in order to distinguish it from the RPGs that were already on the market. DARKNESS FALLS: THE CRUSADE (DFC) featured a built-in player-versus-player (PvP) conflict in which three different teams, called Realms, fought each other for control of magical artifacts, known as Idols. We really liked this concept, which served to keep DFC players hooked on the game — especially because no other online game featured such team-based conflict as a core part of the game design. So, in late 1999, we decided to make a graphical version of DFC. The project was dubbed "Darkness Falls 3D," and we began preliminary work researching client engine and server technology.



MATT FIROR | *Matt has been producing online games since the infancy of the industry. He has produced more than a dozen online games, including SILENT DEATH ONLINE, ALIENS ONLINE, ROLEMASTER: MAGESTORM, and of course, DARK AGE OF CAMELOT. He splits his time between a horse farm in Hunt Valley, Md., and a tiny apartment in Arlington, Va. Matt can be reached at mattf@mythicentertainment.com.*

The NetImmerse graphics engine from NDL proved flexible, stable, and packed with features players expect. Cities, creatures, world objects, and spell effects were created entirely in 3DS Max and exported using NetImmerse's Max plug-in, MaxImmerse.

CAMELOT's engine and client/server technology proved remarkably stable, displaying spell effects and combat animations, parsing system messages and chat, dispensing quests and tasks, and sending countless client/server messages with minimal effect on gameplay.

Right off the bat it was obvious that we had two major factors going in our favor. First, we determined we could use a much-enhanced version of the SPELLBINDER graphics engine as DFC3D's client, just as we were able to use DFC's server code as a platform for the new game's back end. Having such a solid client and server right at the start — with associated client/server messaging — alone saved us at least a year of development. Second, and even more advantageous, DFC's server came with that game's database of objects, monsters, and weapons. Indeed, we went into the CAMELOT project with a huge head start.

We were proceeding along under the DFC3D concept until our president, Mark Jacobs, came up with the idea of basing the game, at least partially, on the Arthurian legends. It was a great idea, since the stories of King Arthur are in the public domain, which meant we could use them with no fear of licensing issues. Of course, because the game was based on the idea that three Realms were in conflict, we quickly came up with the idea of basing the other two Realms on Norse Viking myths and Celtic Irish legends, respectively. Having the myths and legends of three cultures gives CAMELOT the feel of being three games in one, since each Realm has different races, classes, guilds, terrain, and monsters.

Because everyone knows what happened in Arthurian England, we based the game after Arthur's death and developed a back story of conflict among the three Realms. The game was rechristened DARK AGE OF CAMELOT, and around January 2000 we began the project in earnest. A year and a half and untold numbers of Monty Python jokes later, we finished the game.

The initial versions of DARK AGE OF CAMELOT used the rights for a tabletop role-playing game called Rolemaster as a basis for the class and spell systems. Not long into the project, the company that created Rolemaster, Iron Crown Enterprises, filed for bankruptcy, and we lost the rights. This turned out to be good for us, however, because we were no longer required to adhere to a set of rules based on the license — although we did have to scramble for about a week to rename and retune spells and classes and otherwise clear Rolemaster content out of the game.

As a company, Mythic had never before been able to devote all of its resources to any one game — we'd never had a project big enough to pay for it. Because of the sheer size and scope of CAMELOT, we wanted to ensure that everyone at Mythic devoted themselves fully to the project. Doing so required an influx of money, and that's where New York's Abandon Entertainment stepped in. Abandon owns a couple of small companies, each of which specializes in different types of entertainment: a film studio, a web company, and a couple of game content development companies. Abandon wanted to become more involved in game development, so it purchased a minority stake in Mythic. This money allowed us to devote everyone on staff to the CAMELOT project, while also expanding and hiring much-needed programmers and artists. Our spreadsheets showed that we had enough money to support exactly 18 months of development starting from January 2000, giving the project a hard end date of September 2001.

By the summer of 2000, we had nearly our entire team in place. We had about 25 developers working full-time on the project — quite a small number compared to other online RPGs, but our existing technology allowed us to reduce substantially the amount of technical programming staff required. We had five programmers, ten world developers, seven artists, and several other people working on the game.

It all starts with a concept. The troll, a playable race, changed the most over the course of development from a hulking, human-like creature the more mythologically inspired version seen here.

Rob Denton, Mythic's vice president and chief technical brain, was responsible for all client and server programming, as well as the client/server messaging that tied the two together. His input was critical during design discussions, as he could tell us whether an idea would work or not. He immediately categorized features into "doable," "not doable," and the dreaded "on the list," which meant that it could be done, but he wouldn't commit to it. Brian Axelson was in charge of server programming as well as design of the game's combat system — a critical component in a PvP-centric game. Jim Montgomery provided CAMELOT's client interface coding and also designed and coded the game's magical spell system.

CJ Grebb and Lance Robertson led the art team. CJ was responsible for the game's look and feel, while Lance handled figure modeling and animations and managed the team's deadlines. Their team used 3DS Max and Character Studio to create CAMELOT's character and monster models and animations. The character models were technically advanced, as each in-game character has several different parts buried in it that can be turned off and on by the game. So, each model can have a helmet head and a regular head (with hair) without having to load in a new model. Mike Crossmire created the game's spells in 3D Studio, tweaking the NetImmerse system to display animated spells with spectacular results.

The other major group in CAMELOT's development was the world team, led by Colin Hicks. This group was responsible for quests, monster placement, object placement, and just about everything else having to do with creating the world of DARK AGE OF CAMELOT. CAMELOT's economy was designed by Dave Rickey. This economic system ensures that players must continue to spend money as they rise in level, which limits the amount of money that stays in the game. Dave and Mark Jacobs designed CAMELOT's trade skill system, which enables players to make armor, weapons, and other objects in the game — all tied to the economic system.

Among the myriad tasks that I did as a producer (writing, designing, persuading, arguing, and such), my job was to make sure all the teams worked together. I hosted an almost-daily morning meeting (at the wretched hour of 8:30 A.M.) where Colin, Rob, CJ, Lance, and I got together to make sure that we were all on the same page. I was also responsible for maintaining the master game client — all files added to the game had to be given to me, so I could verify they worked and then integrate them with the rest of the game.

For the game's sound and music, we contracted with Womb Music, based in Los Angeles, which had provided music for some of our previous titles. Rik Schaffer, the main guy at Womb, composed a wonderful soundtrack that consisted of several long main scores, as well as many shorter pieces in the style of Celtic, Norse, and old English folk songs, adding a sense of depth and quality to the world.

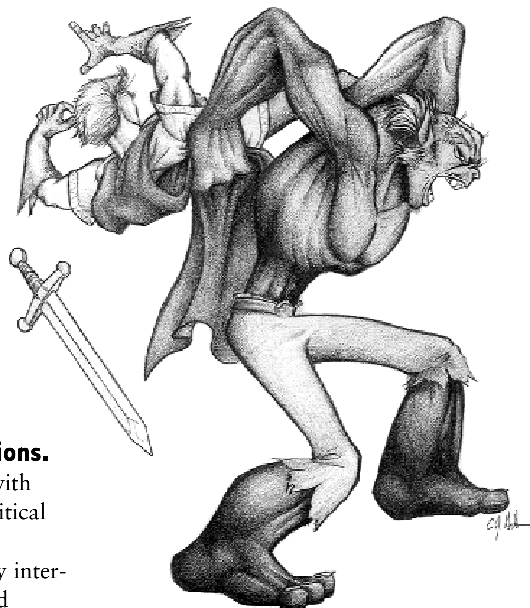
What Went Right

1. Community management/ beta program. From the beginning of the project, we knew we had precious few dollars available for marketing, and that our best chance to capture public attention would be to have a big presence on the various role-playing fan sites around the Internet. One, the Vault Network, provided us with some message board space, a news page, and a couple of moderators, and we were off and running.

We devoted a lot of time over the year and a half that DARK AGE OF CAMELOT was in development to interacting with the future fans of the game. We hired a community relations manager whose sole job was to read different message boards and report back to us what was happening in the community. From the beginning, we took our fans seriously and made many tweaks and additions to the game based on their commentary and ideas.

2. No bureaucracy. Since the founding of Mythic, we have striven to have little bureaucracy. We have no levels, no directors, and few managers. We have a president, a vice president, and a producer. That's it for management, although for CAMELOT we did have to assign a lead world developer and art co-leads, just to streamline the day-to-day processes of the project.

Because of this simple command chain, we experienced no power struggles. We



feel this is the best way to make a solid, cohesive game — a small group controls what the game is and how it is presented to the user. Because of this approach, decisions are made quickly, and features can be implemented without an endless line of approvals and politics.

3. Smart business decisions. Our close relationship with Abandon Entertainment was a critical factor in the success of the game. Abandon's purchase of a minority interest in Mythic ensured that we had enough money to fund the game from start to completion. Abandon's management was smart enough to realize that we knew more about game development than they, so they largely left us to make game-related decisions ourselves. They were involved in the project, of course — some Abandon employees even became avid beta players of the game, even though most had never played an RPG before. Abandon's investment meant that we did not have to rely on any outside influence in designing or creating the game, which means that CAMELOT is wholly ours.

With Abandon teaming with us, Mark Jacobs, our president, decided to take a big chance and wait until the game was almost complete before looking for a distributor. In most cases, game companies seek out publishers, which typically have a hand in the design and production of the game and then distribute the game to the retail chain. With Mark's gamble, we produced the game ourselves (with critical financial help from Abandon and business advice from our business development person, Eugene Evans) and then looked only for a retail distributor. This gamble could have placed us at the end of the project with a great game but no way to get it into the hands of our customers. It all worked out in the end, of course, with Vivendi Universal stepping in and distributing — but on our terms.

4. Sweet serendipity. The CAMELOT project was helped immensely by factors completely out of our control — in other words, blind luck. Several high-profile online RPGs that were slated to launch at about the same time as CAMELOT were either pushed off (SHADOWBANE) or canceled outright (DARK ZION, FALLEN AGE). Also, the week we launched was originally scheduled to be the same week as the launch of WARCRAFT III, which will almost certainly be a huge seller. That project was also delayed, which ensured that CAMELOT launched as the only large-scale game, and the only online RPG, when it debuted on October 9, 2001. This little bit of good fortune gave the game a big initial boost, as there was little direct competition from other new products.

5. The joys of open source software and stability. Long ago, during the development of our early titles, we decided to use Linux wherever possible as our server back-end OS, and we kept to this same practice when creating DARK AGE OF CAMELOT. We have extensive Linux experience in-house, and it made sense for us to stay with a platform that we knew could handle the task and also was, well, free.

Because running CAMELOT would require a considerable amount of data

Balancing the races and classes for effective and challenging player-versus-player combat became one of Mythic's greatest challenges.

It was essential to provide players with plenty of player-versus-environment conflict, such as with the forest giant seen here.

A look at the final version of the Troll. Every race within each Realm was designed to wear the same clothing, so the chain mail seen on this troll had to work equally well on the more diminutive Dwarven race.

In addition to designing CAMELOT's many outdoor areas, Mythic's world development team had to populate those areas with interesting encounters and dynamic quests — no small task, considering they had not one but three distinct Realms to accommodate, as well as a finite amount of creatures available to them. Work on this content is ongoing, with new updates added to the game on a regular basis.

management, we initially planned on using Oracle to store account and character information. However, Oracle's quoted license fee of more than $900,000 quickly removed them from contention. Once we got over our shock and amusement at Oracle's pricing, we turned to a Linux-based freeware solution, MySQL, to manage CAMELOT's data storage, which so far has worked admirably.

Everyone developing games should at least investigate open source solutions for their servers. It's saved us a pile of money and has been stable and reliable. In fact, prior to CAMELOT's launch, it was axiomatic that MMORPGs were unstable and prone to crashing during their first month or so. From the outset, we were determined to buck this trend. We co-located our servers directly at UUNET, on the network backbone, which ensured a wide network pipe to the Internet. With this Internet connection, we can increase our bandwidth with just a few hours' notice to UUNET.

With the combination of reliable server code and a stable Internet connection—all running on open source software — CAMELOT went live on October 9, 2001, with virtually no problems. That first night, the game went down for about an hour and a half due to a database configuration problem, but since then, the game has been remarkably solid and stable. As of this writing, it hasn't been down due to server error for more than a few minutes ever since the first night.

What Went Wrong

1. Development of customer service tools. We really tried to avoid the customer service problems that are characteristic of some recently launched online games. One of the most important factors in keeping customer service reasonably effective was a smooth launch. Obviously, giving players fewer problems results in fewer calls to customer support. We did an excellent job with the launch — it went very smoothly. However, we could have better foreseen other parts of our customer service plans.

First, we had a lot more players in the first week after CAMELOT went live than we ever could have forecast — 51,000 boxes were sold in the first four days alone. Our forecast numbers called for a much smaller number, and we hired our customer service staff based on this smaller number. Also, we put off creating customer service tools until much too late in the development cycle — some had yet to be developed when the game went live. These missing tools really hurt the customer service staff and added to the time it took to help each player with in-game problems. Eventually, wait times became much too long, and customer support as a whole suffered because of it. As I write, we still are trying to work ourselves out of this hole.

2. Lack of a cohesive marketing plan. We went into the CAMELOT project with a lot of experience in developing software, but no real experience in creating a marketing plan. We got

a lot of help with advertising from Abandon Entertainment, but there was no overall project plan. Basically, we took out ads in magazines that we thought were important and tried to keep on top of the Internet community. We didn't regularly issue press releases nor attempt to do a press tour or invite reporters to the Mythic offices to show off the game.

It's difficult to gauge just how much this hurt us. Our focus on Internet marketing gave us strong support among fans of the genre, but our lack of commercial marketing kept our company profile low, and we never received much mainstream media coverage because of it. Fortunately, we made up for our slow start, and then some, by our successful presence at E3. Abandon funded, designed, and staffed a large booth for us at the show, complete with medieval motif and lots of giveaways.

3. O Dungeons and Cities, where art thou? The first major update we made to CAMELOT's graphics engine to differentiate it from SPELLBINDER was to put in the rolling terrain system that makes the world so lifelike. We spent a long time making the outdoor areas of the game beautiful and well stocked with monster encounters. The ease with which we did this gave us a false sense of security when it came to developing our dungeon/city technology.

Creatures were modeled and mapped using 3DS Max and animated with Character Studio. Rumors that this zombie is a portrait of the producer after too many meetings are totally unfounded.

These areas in the game required a large number of models and characters in a much smaller space than the outdoor terrain, so creating dungeons and cities proved to be a much more difficult job than we thought. Because we put off doing the technical designs for the interior spaces for so long, in the end we simply didn't get enough of them done. The game launched with only three capital cities (one per Realm) and about 15 dungeons.

4. We have a great game but no servers! In a great "Why didn't they tell us about this in college?" situation, we went into the final months of the project with no credit rating. Mythic Entertainment has been around for a long time, but we simply hadn't ever borrowed any money, and so we didn't have a credit history. This turned out to be a problem when we went out to lease our servers from Dell and were flatly denied. We pointed out that we had plenty of money in the bank, but to no avail. Dell simply wouldn't lease us the computers until we had a credit history. In the end, we were forced to purchase the servers outright from Dell, which obviously had a much greater impact on our bottom line.

5. Postrelease fan communication. As good as our communication with CAMELOT's fan base was during the game's design and beta periods, it began to suffer soon after the game's release. The community simply grew too large to communicate with in the manner we had during beta, when we simply went out to Internet message boards and posted our thoughts and plans. With the game live, it was obvious we needed a much more coherent way to communicate with our fans, one that would not send them to numerous different fan sites to sift through literally thousands of messages.

This situation grew into a big problem when players became extremely frustrated by what they perceived as a lack of communication from us. About six weeks after release, we realized that we needed to create our own web site to publish information about the game: release notes, plan files, server status, Realm War status, and many other little things that we knew but our players didn't. This web site, dubbed "Camelot Herald," launched the following week and so far has been a great success. Fans of the game can now go to one web site to get all the information about the game in one place and with no interference.

For the Ages

It was a great pleasure to create DARK AGE OF CAMELOT, as it is the first big title that Mythic Entertainment has ever worked on. It was a wonderful thrill to see our names on top of the best-seller lists for those couple of weeks in October 2001, and we hope to be working on the game for a long time to come. As long as players are interested in playing the game, we'll be there adding content and updating it.

The Envelope, Please

Let's imagine games are an art form. I know, I know — for many of us in contact with the so-called real arts, the notion sounds pretentious. It also makes developers who are former computer science majors edgy, because it challenges assumptions that games are founded upon technology. Still, it's a useful concept. It's especially useful when we start to think about the mediocre state of our profession, and about ways to elevate our aims, aspirations, and attitudes.

Art is what people accomplish when they don't quite know what to do, when the lines on the road map are faint, when the formula is vague, when the product of their labors is new and unique. Sound familiar? This is the everyday challenge facing game developers: create something new and unique. Incredibly, we often succeed. The real problem is, new and unique isn't enough — most of us also want to build games that are actually *good*. Good by any standard. Good today, better than yesterday, and worthy enough for tomorrow. Good even when we can't exactly define what "good" means.

How can we focus our energies on such a lofty and elusive goal? It's tough enough to focus on shipping our next title. The best method I know comes directly from some of those "real" arts: the annual round of awards when movie, television, and music academies honor their members' achievements of the previous year.

I believe that awards are an inspiration to all of us — whether we're ever nominated or not. Seeing our colleagues honored raises our sights and ambition above the petty requirements of the marketplace, and also above the dismal recognition that comes from what passes for a trade press in our business. Awards mark the framework in which a consequential meta-discussion about excellence takes place among game developers — buzz translated into votes.

The process is already in place, and while some newer awards programs try to improve on the problems of the older ones, they nevertheless generate struggle and controversy. Should we honor titles or people? Developers or publishers? Should we accept sponsorship? Should we control the nominating process? Should we aim for a marketable entertainment package with our award ceremonies? These issues are important. Establishing a firm basis for our awards will contribute to industry growth and maturity.

I hereby cast a vote in favor of maximum exposure. Awards should discover and celebrate as many of the arts and crafts of game development as possible. We need to spread far and wide the idea that individual human beings are responsible for the games we play. We may spin idle dreams about theoretical possibilities, but what spurs us to action are real achievements, against all odds, by real people toiling in the real world.

continued on page 55

Illustration by Deborah Stephens

continued from page 56

If we honor by title, then we risk award sweeps, as one fad or another dominates in any given year. If we honor by artist and engineer, we stand a chance of delving beneath the surface far enough to acknowledge important, if less trendy, work. It doesn't necessarily follow, for example, that the best game of any year has the best sound, or the finest animation, or the most capable engine.

I cast another vote for simple integrity. Awards should be as free from politics as possible — otherwise they don't mean much. Publishers, manufacturers, and allied companies have their PR machines, their marketing agendas, their bottom lines. Self-interest is built into their charters of incorporation. Awards should not contribute to their further aggrandizement, whether by naming them as recipients or by allowing them to sponsor our ceremonies.

To further ensure community confidence in our awards, we also need to improve our procedures. There's a natural tension between art as craft and art as experience. Here, Hollywood seems to have hit upon a satisfying compromise: Oscar nominations are made by peers, and the final awards are voted upon by all. We should follow this practice. Level designers are the only developers qualified to identify the best levels, for example, yet the rest of us can readily judge the fun factor among selected nominees.

If we work at it, the result will not only be better games, but clout. Hollywood, home of the rudest pop entertainment, has become immune to unfair pressure simply by declaring, through its web of awards, that movies are an art form. Establishing our own well-conceived awards should help protect us against the slings and arrows of outraged congressmen and social busybodies who imagine that good art is like good nutrition — the five food groups of character formation, as it were.

It took decades for the Oscars to become the show-biz phenomenon they now are. Yet Frank Capra's win for directing 1934's *It Happened One Night* is as well remembered as Steven Spielberg's award for 1998's *Saving Private Ryan* because the process was solid from the beginning. Similarly, it may take a while for game awards to acquire public cachet. But to developers, those honored and those voting, the benefits are immediate and lasting.

HAL BARWOOD | *Hal is a project leader at LucasArts, where he is working on a new console title. He is also a member of the Academy of Motion Picture Arts and Sciences. He was given a Spotlight Award for INDIANA JONES AND THE FATE OF ATLANTIS at GDC many long years ago.*