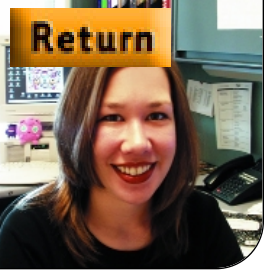
gd

GAME DEVELOPER MAGAZINE

MARCH 2002

UBM Tech



GAME PLAN

LETTER FROM THE EDITOR

Avast, Ye Mateyz

The issue of piracy has plagued the software industry since its earliest days. Recent efforts by the FBI and the U.S. Customs Service (Operations "Buccaneer," "Bandwidth," and "Digital Piratez") to crack down on software piracy rings has rekindled many longstanding debates among developers about what can and should be done about the problem.

An all-too-prevalent sentiment among game developers is that piracy is ultimately the publisher's problem. Established, successful studios enjoying generous royalties are less likely to agree with such a viewpoint, but for younger, unproven studios lacking a royalty plan, what real motive exists to devote precious man-hours to devising and implementing anti-cracking technology just to put a few more dollars in the publisher's pockets?

As we know, the most critical point in a game's life cycle in terms of sales momentum is the period just after a game's release. Staving off the (inevitable) crack for even a few weeks can bolster total sales figures for a given title — a number a prospective publisher will be very interested in when you go shopping for your next title.

Prevailing wisdom would suggest that the most sound argument against software piracy is that — like insider trading or marrying your sister — it is simply against the law. It doesn't matter whether you're running an international ring intent on directly funneling sales from software makers, or you're a lone kid looking for the newest game, perpetrating what you know to be a crime yet consider to be victimless.

I know there are plenty of people working in this industry that first became interested in programming games for a living in part because of lots of work they did playing heaps (pirated) of games as kids. That's one of the great assets of our industry: its homegrown roots and the sense of intimacy and passion that such a heritage propagates. But how did you feel the first time you were working professionally on a project, working day and night through-out crunch mode, only to see the first cracks of your game start showing up on the Internet three weeks before you went gold? Whether it was rage, pride, or some strange combination of the two, at that point you have to admit that pirating games is neither just the publisher's problem nor a victimless crime, because you are the victim.

New Columns Debut. This issue marks the debut of two long-overdue additions to *Game Developer*'s monthly lineup of regular columns, to better incorporate more information for all of our readers on a monthly basis.

"Better by Design" is our new column on game design, kicked off by veteran game designer Noah Falstein. Noah's got an ingenious plan for the column, to identify and build a set of practical game design rules, or what he calls "The 400 Project" (you'll have to see page 26 to find out why), with an emphasis on practical applicability. Eventually, we want to get lots of experienced game designers sharing rules they have identified over the years, thereby building a database of proven game design rules and techniques, which will enable everyone to make better games.

The second addition is a column on game audio, "Sound Principles." This column aims to educate all game developers on how we can get better soundtracks into and out of games. Topics will range from sound design and music composition to audio programming and production planning. Whether it's created by an in-house studio or a lone contractor, achieving great game audio is in many ways a team effort. George "The Fat Man" Sanger starts things off this month with an article wryly called "The Unimportance of Audio," on page 28.

I hope these two new columns open up exciting new avenues of discussion among all developers, something there's never enough of to go around in a rapidly burgeoning industry such as ours.

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
Everard Strong *estrong@cmp.com*
Production Editor
Olga Zundel *ozundel@cmp.com*
Product Review Editor
Tor Berg *tberg@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@blot-action.com*
Hayden Duvall *hayden@confounding-factor.com*
Noah Falstein *noah@theinspiracy.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
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, Silicon Valley & 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

BPA INTERNATIONAL
Game Developer is BPA approved.

CIRCULATION
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, HR & Communications Leah Landro
Vice President & General Counsel Sandra Grayson
Vice President, Creative Technologies Philip Chapnick

CMP United Business Media

GamaNetwork

WWW.GAMANETWORK.COM

INDUSTRY WATCH

THE BUZZ ABOUT THE GAME BIZ | *daniel huebner*

Arakawa retires. Nintendo of America has announced the retirement of president Minoru Arakawa. 55 year-old Arakawa, son-in-law of Nintendo Co. Ltd. president Hiroshi Yamauchi, has served as Nintendo of America's president since the subsidiary was formed in 1980. Tatsumi Kimishima, currently president of Pokemon U.S.A. and chief financial officer of the Pokemon Co., will succeed Arakawa.

Activision acquires Gray Matter. Activision has announced that it is exercising its option to acquire the remaining 60 percent of Gray Matter Interactive Studios, creators of RETURN TO CASTLE WOLFENSTEIN, for $3.2 million in stock. Activision financed the acquisition with an issue of 133,690 shares of common stock, but the company does not expect the deal to affect earnings or revenue guidance during fiscal 2002 or 2003. Key members of the Gray Matter team, including studio head Drew Markham, have signed employment contracts with Activision and will remain with the studio.

Acclaim posts positive results. Acclaim reported a 61 percent jump in quarterly profits. Revenues for fiscal Q1 increased from $72 million last year to $81 million this year. Profits increased to $17.4 million from $10.8 million in the same period last year. Though the results were in line with analyst expectations, Acclaim raised its guidance for the current quarter as well as the fiscal year.

Acclaim attributed much of the good turn to successful internal titles for next generation consoles; 23 percent of the quarterly revenues came from Gamecube titles, despite the fact that the console shipped just two weeks before the close of the quarter.

The company followed the good financial news by announcing the appointment of a new president and chief operating officer. Acclaim has tapped former NBCi.com president Edmond Sanctis to take on the role and oversee all of the company's North American operations.

Eidos's COMMANDOS 2 failed to capture the flag.

Eidos reports loss. Eidos saw a disappointing 44 percent drop in the quarter ended September 30, forcing the company to report a second-quarter net loss of $16.1 million.

Eidos had banked on two major titles in the holiday run-up, but new versions of WHO WANTS TO BE A MILLIONAIRE? and COMMANDOS didn't live up to the company's expectations. Eidos is hoping for better returns on next-generation console versions of the titles and an upcoming Winter Olympics title.

Eidos also announced that it will change its year-end during the current fiscal year to June 30, to help the company better forecast fiscal year results by putting the sales-heavy Christmas season in the middle of its tracking period.

SEC questioning Take-Two's accounting. A story in *Barron's* magazine reports that the Securities and Exchange Commission has asked videogame publisher Take-Two Interactive to explain some of its accounting procedures. Specifically, the company has been asked to shed light on its accounting of certain impaired assets and one-time charges.

According to *Barron's*, the SEC asked Take-Two's chief financial officer why its books place high carrying values on certain impaired assets, how its one-time charges complied with accounting rules, details on its 90-and 120-day payment terms, and an accounting of cash payments on its receivables.

There was no evidence that the SEC's enforcement division is investigating Take-Two, but the letters suggest concern over its accounting for revenues and expenses, citing researcher John Gavin, who obtained the SEC's letters.

Infogrames makes financial moves. Infogrames is making moves to cut its debt and simplify its shareholding structure. The company timed nearly 25 percent of its outstanding debt through a buyback of more than 2 million convertible bonds. Infogrames also sold 8 million of its own shares in exchange for $50 million worth of convertible bonds issued by its U.S. division (Infogrames Inc.) and $50 million in cash.

Shareholders also made moves to simplify the company's structure with the approval of a plan to merge with its main shareholder, Interactive Partners. Infogrames will pay 15.23 euros per share to buy out Interactive, which owns 19 percent of Infogrames' capital and 30 percent of its voting rights. Both moves could potentially serve to refocus interest on Infogrames as a possible takeover target.

UPCOMING EVENTS CALENDAR

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

PLAYING WITH THE FUTURE: DEVELOPMENT AND DIRECTIONS IN COMPUTER GAMING

ESRC CENTRE FOR RESEARCH ON INNOVATION AND COMPETITION, UNIVERSITY OF MANCHESTER
Manchester, England
April 5–7, 2002
Cost: variable
www.digiplay.org.uk/cfp.php

Darkling Simulations' Darktree 2

by steve theodore

As audiences become more and more sophisticated about computer graphics, fractal noise and procedural wood-grain textures can no longer guarantee gushing reviews. Darkling Simulations' Darktree 2 is the latest edition of the popular tool for artists who want to develop complex, multi-layered shaders that transcend the commonplace of current CG. Darktree 2 offers users a broad palette of procedural textures, many of which will already be familiar to 3D artists. The program's main virtue, however, is the ability to drive the parameters of one procedure with the output of another. By feeding the results of these chains of procedures to the texture channels and shading parameters of an object, Darktree 2 can create sophisticated and sometimes startling effects ranging from highly realistic natural surfaces to animated pyrotechnics to flat-shaded cartoon renderings (Figure 1).

Darkling refers to networks of shading procedures as Darktrees. Darktrees can be exported as plug-in materials, which Darkling calls Simbionts, and used in any of several 3D packages. Simbionts act as rendering proxies, representing Darktree shading networks to the host programs as custom materials. Simbiont plug-ins are available for 3DS Max (3.x and 4.x), Lightwave (6.5 and 7), Animation Master 8.5, and Truespace 5.

Darktrees can also be exported as bitmaps or 2D animations in a wide variety of formats and at any arbitrary resolution.

Making Connections

From a computational point of view, Darktrees are linked series of texturing algorithms: in essence, programs, similar to Renderman shaders. Indeed, programmers may find it useful to understand Darktree 2 as a graphical front-end to a streamlined Renderman-like language. Fortunately for artists, however, the experience of working with Darktree feels nothing like creating code.

Darktrees are represented as flowchart-like diagrams of connections between procedures, analogous to electrical schematics (Figure 2). The nodes in the diagram are procedural components that generate, manipulate, or composite texture and parameter data. The wires transmit colors and parameter values from one procedural to another. Although it takes a good deal of skill to create truly useful shaders, the mechanical aspects of laying out Darktrees are very simple and will be grasped quickly by anyone capable of mastering Visio or CorelDraw.

Darktree users work on a grid of regularly spaced sockets. A single root socket at the left side of the grid contains a Darktree shader, a Swiss army knife superset of the Blinn, anisotropic, and clearcoat shaders found in most 3D packages. To build the Darktree network, users drag and drop icons representing various procedures from a component library onto the sockets in the grid. The node icons contain thumbnail previews, making it easy to see what each node contributes to the tree. Once components are placed on the grid, users can connect inputs and outputs easily by pointing and clicking.

Darktree's GUI is sufficient for the task of creating shaders, though in some ways it lacks refinement. The only important drawback to the interface is how the forced data-typing and the rigidity of the layout can make it awkward to share components between two branches of a Darktree. Since so much of the power of the shading network paradigm lies in its ability to coordinate effects in multiple channels (for example, aligning features in a bump map with those in a color map), anything that inhibits sharing is more than an annoyance. On the balance, however, the interface does a good job of managing the complex task of shader building.

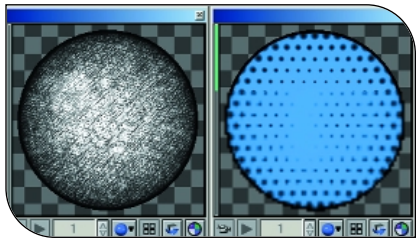


FIGURE 1. Darktree offers flat-shaded textures as well as the familiar photorealistic options.

I've Got Algorithm

Darktree 2 offers an impressive library of more than 100 components for building shaders. The fact that the package includes 10 different kinds of 3D noise textures gives some indication of the depth of the toolset. This large library of procedures is an extremely potent resource for building shaders. The program also includes a large library of complete Darktrees, allowing new users to learn by example, but the absence of a master reference for the components is a disappointment.

The core of the component library is a collection of more than 50 procedural texturing components. Standbys such as fractals, clouds, and grain are all present,

STEVE THEODORE | *Steve is an animator and character designer at Valve Software, where he works on online titles such as COUNTER-STRIKE and the upcoming TEAM FORTRESS 2. He can be reached at stevet@valvesoftware.com.*

excellent
very good
average
disappointing
don't bother

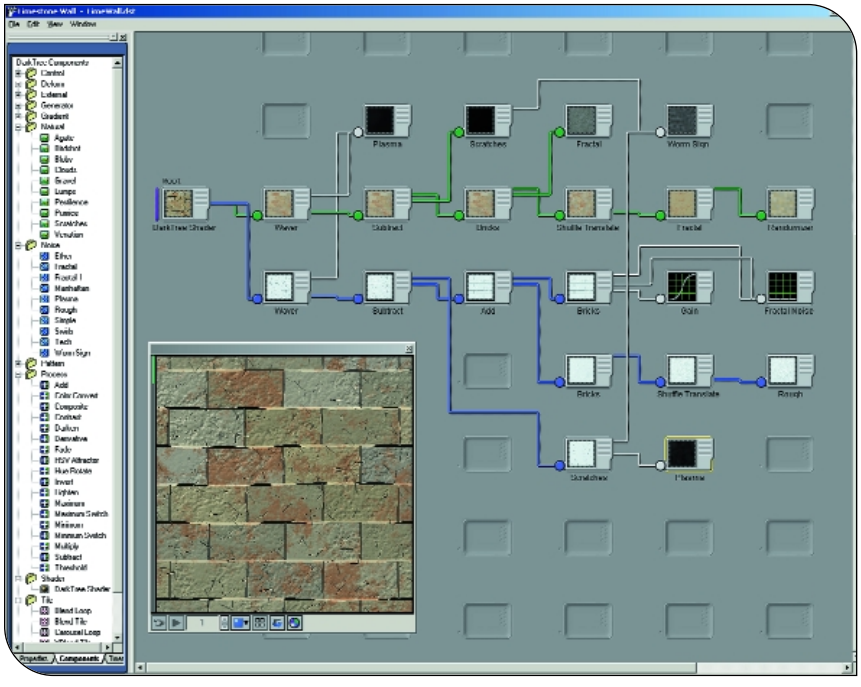


FIGURE 2. Darktree users build procedural shaders with an easy-to-use flowchart-like interface.

as are a number of new procedurals many artists won't have seen before. Unique standouts include Tech, a complex of overlaid trace lines reminiscent of a circuit board; Manhattan, an array of randomized rectangles suitable for metal plating; Venation, which produces a polygonally fractured surface like fractured glass; and Scratches, a 3D array of randomly oriented strokes that can be curved, tapered, and randomly colored.

Artists interested in the far reaches of procedural shader creation will also find a wealth of components for recombining and manipulating textures. Darktree offers a large selection of compositing operators, analogous to Photoshop's level modes. A sizeable library of mathematical functions allows sophisticated control for blending textures and parameter tuning. Modifier nodes can reprocess base texture components for effects ranging from spatial distortion to color shifting and even tile creation. Finally, a number of utility components provide access to external data: anything from imported bitmaps to audio files and animation timers can be used to drive parameters within a Darktree.

Twiddling the Knobs

Every component in a Darktree is highly customizable. The interface for editing individual components is simplistic, but functional. Double-clicking on a component icon opens a modal editing dialog. Feedback is fairly fast when adjusting component parameters, even on relatively anemic machines. However, working with only the single, smallish view swatch in the parameter window can feel like working through a peephole. 3DS Max–style spinners and text boxes handle numerical parameters, and a clever click-drag swatch control lets users edit HSV values gesturally without opening a color picker. Parameters can also be controlled by global variables called Tweaks. These Tweaks are exported when Darktrees are used as plug-in shaders and can be changed or even animated in the host program's material editor. Unfortunately, tweaked parameters in a given component can't be edited from the edit window — Tweaks are treated as properties of the Darktree itself, so altering them requires a trip to the top level of the Darktree.

Who, How, and Why

Darktree 2 is a powerful tool. Artists willing to make a reasonable investment of time in experimentation will be well rewarded. For game developers, the utility of the program depends largely on the task at hand. Cinema teams and sprite artists should definitely consider Darktree 2 unless they already use Maya or Softimage XSI, which include similar functionality out of the box. Modelers creating characters for real-time games, however, are unlikely to find it a compelling purchase: the Darktree shaders are irrelevant to real-time games and without a method of converting solid procedural textures into UVW-mapped textures, the Darktrees are only an elaborate method for producing fill patterns. Level designers who don't work with UV surfaces, on the other hand, should get a lot of value out of rendered Darktree bitmaps. Overall, Darktree is a tool with great potential and is definitely worth investigating.

DARKTREE 2 ★★★★

STATS

DARKLING SIMULATIONS
Los Alamos, N.M.
(505) 672-0640
www.darksim.com

PRICE
$495 ($125 upgrade from version 1)

SYSTEM REQUIREMENTS
Windows 98 or 2000; 500MHz processor; minimum 128MB RAM

PROS

1. Powerful tool for creating custom procedural shaders.
2. Large library of texturing components.
3. Generates high-quality bitmaps.

CONS

1. Needs master reference to shading components.
2. Rigid editing interface.
3. Sharing components between shading channels difficult.

RELIABLE SOFTWARE'S CODE CO-OP 3.2

by johnathan skinner

Code Co-op is a simple-to-use and affordable version-control system by Reliable Software. Geared toward small- to medium-sized projects, the server-less system maintains a full, up-to-date copy of a project database and updates check-ins over a LAN or even through e-mail for remote development (using Microsoft Outlook). This makes it most useful for teams that have people in several different locations or that use sometimes-connected machines such as laptops.

Code Co-op is available for Windows 95/98/NT 4/2000/XP and has a quick and simple installation and setup. It has an easy-to-use GUI, direct integration with Microsoft Visual Studio, and an optional command-line interface. Some of the GUI layout is a little strange, but it has lots of good tool tips and helpful message boxes that make it easy for users to figure out pretty much everything without needing to refer to the documentation. I am impressed by the help docs, though, as they are well written and include lots of clear and relevant screenshots. The tutorial guides you through setting up your first project. Technical support is also available via e-mail, and Reliable responds quickly. The company is also known to post new fixes for bugs within a few days of finding the problem.

Boasting a reasonable price tag of $145 per user, Code Co-op is well suited for a small development team. It also saves you the cost of purchasing a server machine. One really nice thing about the server-less model is that you control when the project database is updated. You know those times when you need to check out and edit a single file, but then discover that someone else has already made a change to it? You'd need to pull down the latest version of the entire project and recompile it just to change a single line. Code Co-op cures this headache by keeping an entire copy of the project database on each development station, allowing team members to update it anytime they're ready.

But the server-less model does have its drawbacks. While Code Co-op does maintain a full version history for both text-based and binary files, a project with hundreds of megabytes of binary resources (such as art or sound files) all under source control is going to fill up everyone's hard drives really quickly. However, for projects with only a few binary files and mostly just code under source control, Code Co-op should work quite well.

Code Co-op's lack of administrative options may make it less than ideal for larger development teams (more than about 10 people using source control). The only permission control available changes a user to an observer (in which case he or she cannot make changes to the project). Typically, a small team wouldn't need anything beyond that, but a larger team often needs more advanced options.

Reliable Software's Code Co-op 3.2 is an excellent version control system for a small development team. A free 31-day full-featured trial version is available for download from Reliable's web site.

★★★½ | Reliable Software
www.relisoft.com

Johnathan Skinner has been a professional game programmer for five years, starting with TETRISPHERE *for Nintendo 64. He is currently working at Relic Entertainment on its upcoming game,* IMPOSSIBLE CREATURES.

HOUSE OF MOVES' DIVA 1.7

by john bunt

It's the day after our big mocap shoot, and I have to fix a run animation wherein the actor's right leg is longer than his left. Unfortunately, when we captured the move, nobody noticed that the actor's right-ankle marker had slid down by two inches. This small error is big enough to create a glaring limp in the run; and even worse, the same problem exists in 50 other files.

All 50 files contain an average of 300 frames. The marker must be adjusted for every single frame on every single axis in relation to every other body marker for every one of their axes. Depending on how many editors assist, the process could take days of tedious work resulting in data that will still look bad.

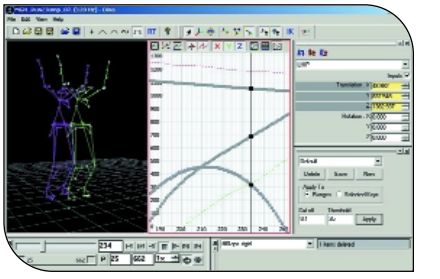


FIGURE 3. The Diva interface displaying second figure data (in purple) in the 3D Perspective view and in the Graph view (dotted line).

Not anymore. After a year of editing motion data with archaic tools that kept me staring at my monitor till my eyes bled, I was introduced to Diva 1.7. Diva is a stand-alone motion capture animation processing tool developed by the mocap pros at House of Moves for their own in-house editing, and now it's being offered as a commercial product.

With Diva, it took me five minutes to write a simple script to reposition the misplaced ankle marker (relative to the foot markers) and batch-process the adjustment on thousands of frames over all 50 files. What normally would have taken days to accomplish, I alone completed in minutes using Diva. Best of all, the data looked great; much better than if I had manually adjusted the marker at every frame.

Built from the ground up to edit motion capture animation, Diva eliminates the frustration of working with older suites of motion capture software that typically ship with mocap hardware systems. It addresses the most common problems of motion capture, such as filtering jittery noise and filling gaps in data. The data-editing tools are extremely accurate, providing several variables for fine-tuning.

Diva also includes a flexible, open-architecture scripting language to handle the unexpected. You don't need to be a programmer to write scripts in Diva, because the help files and tutorials are so

intuitive. Not unlike MEL script for Maya, Diva's HSL scripting language allowed me to customize Diva's operations and assign them to customizable toolbar buttons. Diva also comes equipped with several default scripts to execute most of my daily editing operations.

Right out of the box, I was able to use Diva with my unique data and marker sets. In addition, Diva easily accommodated my pipeline, as it supports every major mocap file format, including marker (.CSM, .C3D, .TRC) and skeletal (Acclaim, .BVH, .HTR). Additionally, Diva exports a proprietary .HDF file format similar to a 3DS Max or Maya scene, containing all objects such as markers, bones, rigid bodies, and solvers.

I was especially impressed with Diva's Second Figure feature (Figure 3), which allowed me to edit data and compare it side-by-side to the original data (the second figure). This took the guesswork out of whether or not I had over-filtered or wrongly edited. I could view the second figure data in Diva's 3D Perspective and Graph views and had the option to restore any original data at any time over a selected range.

There's no doubt that Diva is a versatile program that has made a huge breakthrough in animation. You no longer have to be intimidated by working with motion capture data. Diva's price tag of $3,995 is well justified when you consider how much time and money you'll save by not having to hire extra motion editors to work all those extra hours.

★★★★★ | Diva 1.7 | House of Moves
www.moves.com

John Bunt is the director of motion capture at Blur Studio, an animation studio in Venice, Calif.

PHYSICS FOR GAME DEVELOPERS
BY DAVID M. BOURG

reviewed by jeff lander

As you all know, creating a modern game is a complicated process requiring a dedicated team with a great wealth of expertise. It's no longer sufficient for a programmer to know how to display an image on the screen. Programmers on game projects are required to have knowledge of 3D graphics, advanced artificial intelligence techniques, high-level mathematics, and enough physics to scare an out-of-work aerospace engineer. If your company doesn't have the luxury of being able to hire one of these rocket scientists to create the physics in your game, you may be asked to design the physics system. If you are like me and have forgotten most of your high school and college math and physics lessons, you have some relearning to do.

There's quite a bit of material out there. My oldest physics book is from 1952 and still very useful. However, figuring out how to apply various exercises and examples to game problems can be difficult. Most academic physics books weren't written with the needs of a modern action game in mind.

Physics for Game Developers by David Bourg is designed specifically for the game developer who is creating a physical simulation. The book contains pretty advanced stuff; it's not *Game Physics for Mathematical Imbeciles*. It assumes the reader is pretty comfortable with trigonometry, matrix mathematics, and the basic principles of physics. This base assumption is necessary, since there's quite a bit of ground to cover on the topic.

The book does a very good job of jumping into the basics of Newtonian physics and how it applies to actual game development issues. The physics of particles and rigid bodies make up the bulk of the work. It tackles difficult-to-explain topics such as basic force application, inertia tensors, and collision response in a clear and practical manner.

This foundational section is followed by specific application cases that are directly relevant to game developers. The chapters on projectiles, ships, and cars contain interesting information that programmers will find useful. The sections that follow are on the actual creation of a game simulator, making use of the knowledge in the prior chapters.

Abundant sample code segments also differentiate this title from most physics books. The code in the book is clearly written and well commented, making it immediately useful. The book's web site also hosts sample applications that are referenced in the text. However, these applications are not the book's strong point. While the actual routines from the book are well commented and written in a portable C++ style, the Direct3D applications used to show off the techniques are not nearly as good.

There are some small problems with the book itself, as well. Numerical integration issues are discussed and a variety of approaches are described, but the examples mostly use Euler's method, and the stability problems that can arise are avoided by careful tuning of the simulation parameters. For any actual game situation, numerical stability is a major part of the simulation creation process. As this can be a very frustrating part of the development of a physics system, a more realistic and broader discussion would have been more helpful to readers. While the sections on applications are interesting, they tend to gloss over the issues. For example, vehicle physics is discussed, although connecting multiple rigid bodies into a system is not discussed beyond a brief mention of springs. This information would be critical to creating a realistic driving simulation. Also, the section at the end dealing with mass-and-spring systems is really just a bare overview and not terribly useful for anything other than the most simple of effects.

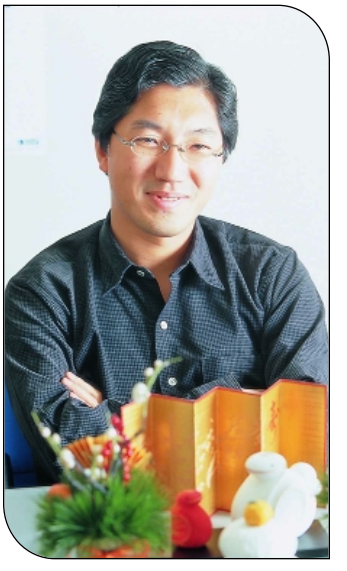
As these are all more advanced issues, I certainly hope that the series will continue into a second volume. For the experienced game developer who is looking to learn about physical simulation, this title will provide a good, solid, and practical foundation.

★★★★ | *Physics for Game Developers* | O'Reilly | www.oreilly.com

Jeff Lander is the founder of Darwin 3D, a graphics programming consultancy on a secluded beach in sunny California. A former Game Developer *columnist, he missed us so much that he's back already. E-mail him at jeffl@darwin3d.com.*

Yuji Naka: Sonic Success

Yuji Naka joined Sega in 1984 as a programmer, and put a new face on gaming with the 1991 debut of his creation, Sonic the Hedgehog. As president and CEO of Sonic Team, a member of the Sega Group, he recently oversaw the development of SONIC ADVENTURE 2 for Gamecube, as well as some of the Dreamcast's most innovative titles, such as PHANTASY STAR ONLINE and that simian rhythm-action classic, SAMBA DE AMIGO. What goes on inside the heads of people who put monkeys and maracas together for the masses? We caught up with Naka-san recently to find out.



SONIIC THE HEDGEHOG creator Yuji Naka

Game Developer. What do you think is the single biggest difference between the Japanese and the Western approach to game design?

Yuji Naka. Game design depends entirely on the gamers who are playing them. I feel that people from Western countries prefer more challenging games than the Japanese do. This affects our approach to game design.

GD. What are some of the fundamental considerations for designing successful, enduring game characters?

YN. I think that the best characters are always born from a necessity in the game. For example, in the SONIC THE HEDGEHOG series, we needed a character that could not only run very fast, but who could also protect itself — thus, Sonic the Hedgehog was born. Another important element in character development is that the game itself also has to be fun to play.

GD. What was the first game you ever worked on? When did you realize that you wanted to develop games for a living?

YN. The first game I ever worked on was GIRL'S GARDEN for the SG-1000 console. [This was a game about a girl in a garden who must pick a flower for a boy, all the while overcoming the challenges of the garden, including getting stung by a bee.] I was in my third year of high school when I decided I wanted to work in the game development business. But honestly, at the time, I did not know that I would ever be able to develop games.

GD. How do you determine what you think your game-playing audience will find fun and engaging? Do you do focus testing, rely on your experience, or just make what you know you would have fun playing?

YN. Although there are a number of factors, including consumer tests, which contribute to our decision to create a game, I believe the best tactic is always to believe in your idea and follow your own instinct. When you do that, everything else will adjust accordingly and fall into place. It has been my experience that this approach usually results in the most successful games.

GD. Besides the games that Sonic Team works on, what sorts of games interest you the most right now, and how do they influence your work?

YN. I am currently most interested in titles from Nintendo. Although every gaming experience has its influence on the player, our aim is to create truly original works that are unlike any other game.

GD. SAMBA DE AMIGO captured many people's imaginations. Actually, I don't know how we made it so long without any games that combined monkeys and maracas. How much interest do you have in exploring alternative input devices for games in the future? Are they nothing more than the occasional novelty for a single game, or are they the real future of immersive interactive entertainment?

YN. I believe that the fun elements in games can be greatly enhanced with the use of various peripherals, such as the maracas in SAMBA DE AMIGO. In the future, I plan to create more interesting peripherals, and I am currently considering various possibilities that I can't discuss at this time.

GD. PHANTASY STAR ONLINE surprised a lot of people with how well executed a real-time multiplayer online console game could be. What do you think people will be doing with game consoles and their Internet connections in five or 10 years?

YN. In five to 10 years, I believe that about half of all games will be connected to the network without having to set anything up — it will be seamless. In five to 10 years, Internet gameplay will largely depend on game creators and whether or not the online component is integral to the gaming experience. I would like to create games that explore new ways of using the network.

GD. What did you think the first time you saw Sonic on a non-Sega console?p

YN. Sonic the Hedgehog celebrated his 10th anniversary this year, and Sonic has been saying that he wants to run on new hardware platforms. Thus, he started to run on Nintendo Gamecube the year of his 10th anniversary. I feel very deep emotion about this. I consider it to be one of the most historical events in the game industry.

Hacking Quaternions

Quaternions are a nifty way to represent rotations in 3D space. You can find many introductions to quaternions out there on the Internet, so I'm going to assume you know the basics. For a refresher, see the papers by Shoemake or Eberly in For More Information. In this article we will look closely at the tasks of quaternion interpolation and normalization, and we'll develop some good tricks.

Interpolation

When game programmers want to interpolate between quaternions, they tend to copy Ken Shoemake's code without really understanding it (hey, that's what I did at first!). Ken uses a function called slerp that walks along the unit sphere in four-dimensional space from one quaternion to the other. Because it's navigating a sphere, it involves a fair amount of trigonometry, and is correspondingly slow.

Lacking a strong grasp of quaternions, most game developers just accept this: slerp is slow, and if you want something faster, maybe you should go back to Euler angles and all their nastiness. But the situation is not so bad. There's a cheap approximation to slerp that will work in most cases, and is so brain-dead simple and fast that it's shocking. Shocking, I tell you.

What Slerp Does

Slerp is desirable because of two main properties; any approximation we formulate would ideally have the same properties. The first, and perhaps most important, is that slerp produces the shortest path between the two orientations on that unit sphere in 4D; this is equivalent to finding the "minimum torque" rotation in 3D space, which you can think of as the smoothest transition between two orientations. The second property of slerp is that it travels this path at a constant speed, which basically means you have full control over the nature of the transition. (If you want to add some style, like starting slowly and then speeding up, you can just spline your time parameter before feeding it into slerp.)

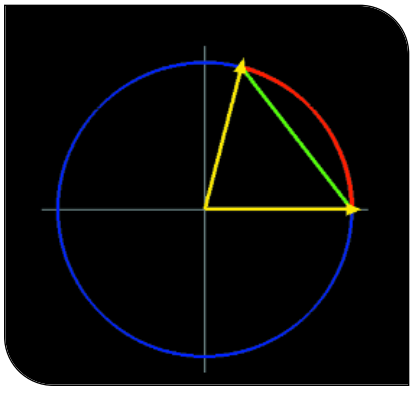


FIGURE 1. A two-dimensional picture of quaternion interpolation. The blue circle is the unit sphere; the two yellow vectors are the quaternions. The red arc represents the path traveled by slerp; the green chord shows the path taken by linear interpolation.

So here's our approximation: linearly interpolate the two quaternions componentwise. That is, if t is your time parameter from 0 to 1, then $x = x_0 + t(x_1 - x_0)$, and similarly for y, z, and w.

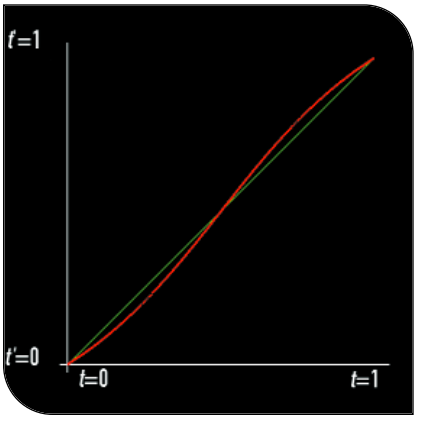
One might wonder how that could possibly be a worthwhile interpolation when the right answer is so much more complicated. Let's take a look at why that is.

Figure 1 shows a two-dimensional version of quaternion interpolation. Slerp walks around the edge of the unit circle, which is what we want. Linear interpolation results in a chord that cuts inside the circle. But here's the thing to realize: Normalizing all the points along the chord stretches them out to unit length, so that they lie along the slerp path. In other words, if you linearly interpolate two quaternions from $t = 0$ to $t = 1$ and then normalize the result, you get the same minimal-torque transition that slerp would have given you.

The linear algebra way to see this is that both the great circle and the chord lie in Span(q_0, q_1), which is a 2D subspace of the 4D embedding space. Adding the constraint that Length(Interpolate(q_0, q_1, t)) = 1 reduces the dimensionality to one, so both paths must lie along the same circle. And both forms of interpolation produce only a continuous path of points between q_0 and q_1, so they must be the same.

If q_0 and q_1 lie on opposing points of the sphere, the chord will pass through the origin and normalization will be undefined. But that's O.K. — unless you're doing something wacky, you don't

JONATHAN BLOW | *Jon (jon@bolt-action.com) would like you to know that the Slimelight in London is a horrible nightclub that really isn't any fun.*



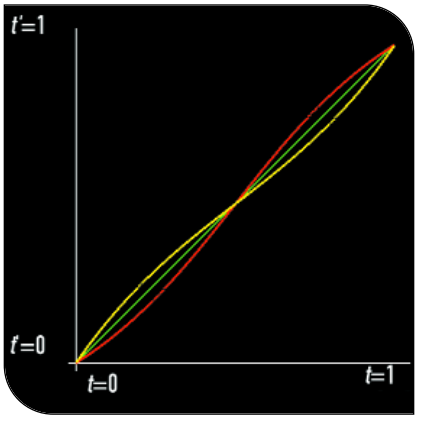


FIGURE 2 (left). Worst case of lerp speed variation. The green line represents the ideal result produced by slerp; the red line represents the distorted result produced by lerp. The error between these two functions should be measured vertically, so they are more different than they may appear at first. FIGURE 3 (right). The compensating cubic spline, $k = 0.45$, shown in yellow atop the graph of Figure 2.

want your quaternions to be more than 90 degrees apart in the first place (because every rotation has two quaternion representations on the unit sphere, and you want to pick the closest ones to interpolate between). So the normalization will always be well defined.

Thus the normalized linear interpolation and the slerp both trace out the same path. There is a difference between them, though: they travel at differing speeds. The linear interpolation will move quickly at the endpoints and slowly in the middle. Figure 2 shows a graph of the worst case, 90 degrees.

The function graphed in Figure 2 is roughly

$$\tan^{-1}\frac{t\sin\alpha}{1+t(\cos\alpha-1)}$$

where α is the original angle between the two quaternions. I figured this out just by drawing a 2D graph like Figure 1, where one of my vectors is the x-axis (1, 0) and the other one is (cos α, sin α). Then I just wrote an expression for linearly interpolating between them by t, and then finding the resulting angle by $\tan^{-1}$. This rather simplistic approach is valid for two reasons. First, since all the action happens in Span(q_0, q_1), we can just take that 2D cross section out of 4D space; studying it in isolation, we see the entirety of what is happening. Second, on the resulting 2D unit circle, because the set of all possibilities for the two unit vectors is redundant by rotational symmetry, we can choose one of the vectors to be anything we like; I chose (1, 0) to simplify the math.

Casey Muratori of RAD Game Tools is the first person I know of who considered linear interpolation of quaternions as a serious option. He investigated numerically and found linear interpolation, when properly employed, to be quite worthwhile. Casey has eradicated all slerps from his code for Granny 2.

Augmenting Linear Interpolation

The linear interpolation is monotonic from q_1 to q_2, so if you are doing an application where you're binary searching for a result that satisfies some constraint, using the linear interpolation works just fine. If your quaternions are very close together (less than 30 degrees, say), as you have when playing back a series of time-sampled animation data, linear interpolation works fine. And if you have some number of different character poses (like an enemy pointing a gun in several different directions), and you want to mix them based on a blending parameter, linear interpolation probably works fine.

Linear interpolation won't work if you need precise speed control and wide interpolation angles. But maybe we can fix that.

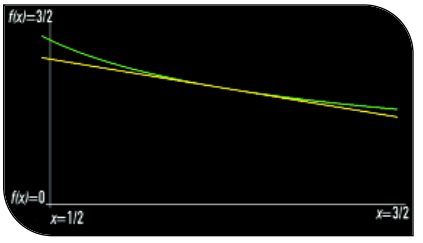
Perhaps we can make a spline that cancels most of the speed distortion. Looking at Figure 2, can we concoct a function that, when multiplied against the curve, causes it to lie much closer to the ideal line? The way I chose to visualize this was with a cubic spline that tries to pull the distortion function onto the diagonal. Figure 3 shows a cubic spline with the equation $y = 2kt^3 - 3kt^2 + (1 + k)t$, where the tuning parameter $k = 0.45$ has been graphed against the plot of Figure 2.

Because both the distortion curve and our compensating spline have an average value of t and are approximately complementary, when we multiply them together we get a function that is approximately $g(t) = t^2$. We want $g(t) = t$, so we'll divide the cubic spline by t. Fortunately, since the spline passes through the origin, it has no d coefficient; so dividing by t just turns it into a quadratic curve: $y = 2kt^2 - 3kt + 1 + k$.

So now, if we're linearly interpolating two splines that are 90 degrees apart, we find $t' = 2kt^2 - 3kt + 1 + k$, and use t' as our interpolation parameter. We get something very close to constant-speed interpolation (I will quantify how close in a little bit). However, if we reduce the angle between the input quaternions, we get something that's less accurate than the original t.

That's because, by defining its slope at $t = 0$ and $t = 1$, I concocted this spline specifically for the worst-case scenario. That's where the k parameter comes in: it's a slope-control mechanism. To get this spline to compensate for distortion across the full range of quaternion input angles, we want to adjust the tuning parameter as some easily computable function of the angle between the two quaternions.

Well, taking the dot product of two quaternions gives us cos α, the cosine of the angle between them. I started playing around with simple functions of cos α until I found something reasonable. Basically, we want a function that is 1 when cos α = 0, and that is near 0 when cos α = 1. After some experimentation I



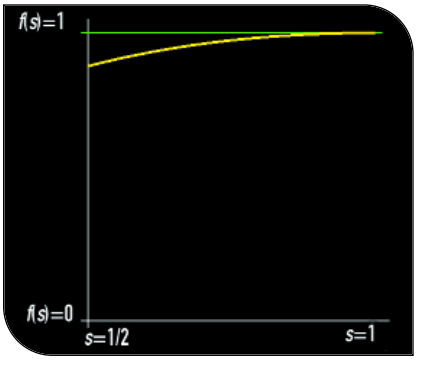


FIGURE 4 (top). In green, the function $f(x) = 1/\sqrt{x}$ in yellow, its tangent line at $x = 1$.
FIGURE 5 (bottom). The length of an approximately normalized vector (yellow), versus the squared length of the input, when using the naive tangent line approximation. The green line indicates the ideal result.

landed on $k = 0.45(1 - s \cos \alpha)^2$, where $s = 0.9$ for now. To cursory visual inspection, this gives pretty good results across the full range of α from 0 to 90 degrees.

These numbers are in the right neighborhood, but because I just made them up, they're not going to be as close as we can get. So I wrote some code to do hill-climbing least-squares minimization. The initial distortion function has an RMS error of about 1.6×10^{-2} when averaged over all interpolation sizes (the worst case, graphed in Figure 2, has an RMS error of 3.234×10^{-2}). The minimizer gave me the following values: $k = 0.5069269$, $s = 0.7878088$, yielding an overall error of 2.07×10^{-3}, which is about eight times lower than we'd started with (see Listing 1).

But while I had been aligning things by eye, I noticed that if I gave k a high value, I got results that were close to exact from $t = 0$ to $t = 0.5$, but diverged after $t = 0.5$. So I wrote an interpolator that only needs to evaluate t from 0 to 0.5. If you pass in a t higher than 0.5, it just swaps the endpoints of interpolation. Running the optimizer on this, I got $k = 0.5855064$, $s = 0.8228677$, overall error 5.85×10^{-4} — a reduction of more than 27 from the original. We incur another small cost to gain this accuracy, an extra `if` statement.

You can probably do better than these numbers; my methods were ad hoc, and there are many possibilities I haven't explored. I should also give a few warnings. For example, the `if` statement I just mentioned introduces a slight discontinuity at $t = 0.5$; you can fix this discontinuity by shifting the midpoint away from 0.5, but this wasn't important for my needs.

So we can interpolate pretty quickly now, but we end up with non-unit quaternions. We probably want unit quaternions, so how do we normalize without doing a really slow inverse square root operation?

Normalization

To normalize any vector, quaternions included, we want to divide the vector by its length. The squared length of some vector v is cheap to compute — it's v·v — so we need to obtain $1/\sqrt{v \cdot v}$ and multiply the vector by that. Division and square-rooting are pretty expensive, though.

We can compute a fast $1/\sqrt{x}$ by using a tangent-line approximation to the function. This is like a really simple one-step Newton-Raphson iteration, and by tuning it for our specific case, we can achieve high accuracy for cheap. (A Newton-Raphson iteration is how specialized instruction sets like 3DNow and SSE compute fast inverse square root).

The basic idea is that we graph the function $1/\sqrt{x}$, locate some neighborhood that we're interested in, and pretend that the function is linear there. A linear function is cheap to evaluate.

So, we want to approximate $f(x) = 1\sqrt{x}$. We are interested in vectors whose lengths are somewhere near 1, meaning $f(x) = 1$, which means $x = 1$. So we are going to focus on the neighborhood $x = 1$, as you see in Figure 4. To get the line, we just take the derivative of f,

$$f'(x) = -\frac{1}{2}x^{-\frac{3}{2}}$$

, and evaluate it at 1:

$$f'(1) = -\frac{1}{2}$$

An equation that says "locally, a function is approximately its value at some point plus its first derivative extrapolated over distance" is:

$$f(x + \Delta x) \approx f(x) + \Delta x f'(x)$$

We evaluate this at $x = 1$ to get

$$f(1 + \Delta x) \approx f(1) + \Delta x f'(1) = 1 - \frac{1}{2}\Delta x$$

Now for the last trick: we want to represent the squared length of our input vector, which we'll call s, as a value in the neighborhood of 1, so we can plug it

LISTING 1. A function that splines t to compensate for the distortion induced by lerping.

```
float correction(float t, double alpha, double k, double attenuation) {
    double factor = 1 - attenuation * cos(alpha);
    factor *= factor;
    k *= factor;

    float b = 2 * k;
    float c = -3 * k;
    float d = 1 + k;

    double t_prime = t * (b*t + c) + d;

    return t_prime;
}
```

into our new linear function. We say $s = 1 + \Delta x$, and thus $\Delta x = s - 1$.

That is all we need. When we plug $\Delta x = s - 1$ into our approximation, we get

$$f(1+s-1) \approx 1 - \frac{1}{2}(s-1)$$

Simplified, this says:

$$f(s) \approx \frac{1}{2}(3-s)$$

For as wide of a neighborhood as the inverse square root is well approximated by a tangent line, this extremely fast computation will give us the factor to normalize a vector. Figure 5 graphs the vector lengths we get when we use this computation to normalize. As long as we start with a vector whose length is near 1, we get results that are fairly accurate.

For some applications, accuracy in a narrow range is all we need. If you are reconstructing quaternions from splines, as one might do in a skeletal animation system that stores animation data in a small memory footprint, you can ensure a maximum length deviation during the spline-fitting process (inserting extra keyframes to alleviate any problems). Then at run time you just evaluate the splines and pump the coefficients into this one-step normalizer, and you can be assured that the results are good.

On the other hand, this isn't good enough to use blindly on the results of quaternion linear interpolation. We can see that, during our worst-case interpolation from (1, 0) to (0, 1), the closest we get to the origin is (1/2, 1/2), which gives us a squared vector length $s = 1/2$. So for good results after lerping, we need a fast normalizer that produces good results all the way through the interval from $s = 1/2$ to $s = 1$.

Retuning the Tangent Line Approximation

When we linearly interpolate quaternions, we get a chord that cuts through the unit sphere; that is, the resulting length is always less than 1. So we don't need our linear approximation to be accurate above 1. We can, in effect, slide the graph of Figure 5 to the left a little bit, making our approximation more effective for shorter vectors.

Also, if we are going to permit some small amount of error ε in our result, it probably makes sense to allow results in the range $1 \pm \varepsilon$, instead of just $1 - \varepsilon$ as in Figure 4. So we can scale the approximation by some small factor. This roughly doubles the zone of good results.

But this still doesn't cover the full range from 1/2 to 1. A simple solution would be to just check the value of s, and if it is too low, just compute the answer the slow way. For most applications, wide-angle interpolations will be extremely rare, so the speed hit will be small. But if you need to be faster than that, there are some hackish things we can do.

I wrote some code that repeatedly applies the fast normalization, tuned by some optimization parameters, in order to achieve the least error across the interval we are interested in. Running the numerical optimizer on this yields $x_{offset} = 0.959066$, scale = 1.000311, and a root-mean-square error of 2.15×10^{-4}. This loop only iterates at most three times over the interval we care about, so you can rephrase the loop as a small series of nested `if` statements, which are mostly never descended into (see Listing 2).

LISTING 2. A fast normalizer.

```
inline float isqrt_approx_in_neighborhood(float s) {
    const float NEIGHBORHOOD = 0.959066;
    const float SCALE = 1.000311;
    const float ADDITIVE_CONSTANT = SCALE / sqrt(NEIGHBORHOOD);
    const float FACTOR = SCALE * (-0.5 / (NEIGHBORHOOD * sqrt(NEIGHBORHOOD)));

    return ADDITIVE_CONSTANT + FACTOR * (s - NEIGHBORHOOD);
}

inline void fast_normalize(float vector[3]) {
    float s = vector[0]*vector[0] + vector[1]*vector[1] + vector[2]*vector[2];
    float k = isqrt_approx_in_neighborhood(s);

    if (s < 0.83042395) {
        k *= isqrt_approx_in_neighborhood(s);

        if (s < 0.30174562) {
            k *= isqrt_approx_in_neighborhood(s);
        }
    }

    vector[0] *= k;
    vector[1] *= k;
    vector[2] *= k;
}
```

Sample Code

This month's sample code implements fast linear interpolation and renormalization, as well as the numerical optimization code that computes the best parameters. Download it from www.gdmag.com.

FOR MORE INFORMATION

Eberly. David. "Quaternion Algebra and Calculus." www.magic-software.com/Documentation/quat.pdf

Shoemake. Ken. "Animating Rotation with Quaternion Curves." *Computer Graphics* Vol. 19, No. 3 (July 1985).

Focus

An industry friend of mine recently pointed out that some projects appear to have everything they needed to produce a visually stunning game, but somehow fail to deliver in the end. The walls in the faultlessly rendered dungeons ooze with displacement-mapped translucent slime; the hair on your character's chest moves realistically, as the icy winds of the Nether Kingdoms blow around him. The Warrior Princess who's about to crush you like a squealing man-worm has buttock movement so perfect in its execution that even Jennifer Lopez would be reduced to tears. Despite all of these features, however, the game world is missing something, and is somehow less than the sum of its beautifully crafted parts. Why? What magical ingredient has been overlooked? Well, needless to say, there are lots of possibilities, but one area often ignored and seldom addressed explicitly is that of focus.

FIGURE 1. Chameleon's eye.

What do I mean by focus? I'm not referring to the kind of focus that relates to lenses, focal length, depth of field, and so on. While these effects are great in prerendered work, they don't really have a place in the real-time rendered world of games at present. To explain what I mean by focus, it is worth considering some of the ways in which we make sense of what we see.

Chameleons Have It Easy

Not only does a chameleon have the ability to adapt the pigmentation in its skin to match its surroundings, but it can also move each of its eyes independently, which gives a chameleon the ability to look at more than one thing at a time. This, you may have noticed, is in stark contrast to the way we humans see things. We are limited to analyzing a scene one area at a time.

While you're sitting there reading this article, you may want to pause for a second and glance around at your surroundings. Go ahead — it'll help me illustrate something useful. For the sake of argument, let's say that you're reading this article in a crowded cell somewhere in Central America. (I don't care how you got there, I'm just glad your cellmates subscribe to *Game Developer*.) There's a prison guard sitting at a table to your left, building a church out of spent matches. To your right, a large hairy man is asleep on the top bunk; as he snores, his impressive moustache rises and falls to the rhythm of his breathing.

With just the briefest of glances, you have been able to establish many details about your location. Looking straight ahead, you are conscious of most everything that's in front of you, as your field of view covers almost 180 degrees. But this apparent visual awareness is deceptive.

Look up again, this time straight ahead. You will most likely perceive a large amount of detail that includes most of the things in front of you, but how much of what you're seeing is in sharp focus? The answer: Very little. Staring at the center of this page, you'll notice that all but a few words are actually out of focus. Peripherally, you can see the shapes of the paragraphs, and you'll even be able to perceive the size and spacing of the words, but only the words immediately surrounding your point of focus will be completely discernable. As this example illustrates, humans have a very small area of high visual acuity.

So, going back to our jail: what you may have thought was a brief glance to your left actually consisted of a rapid set of eye movements, taking in the guard, his table, and his spent-match church. When you looked to the right, you established that the sleeping man was hairy, that he had a significant moustache, and that said facial hair moved in time with his breathing.

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.*

FIGURE 2. A girl's face.

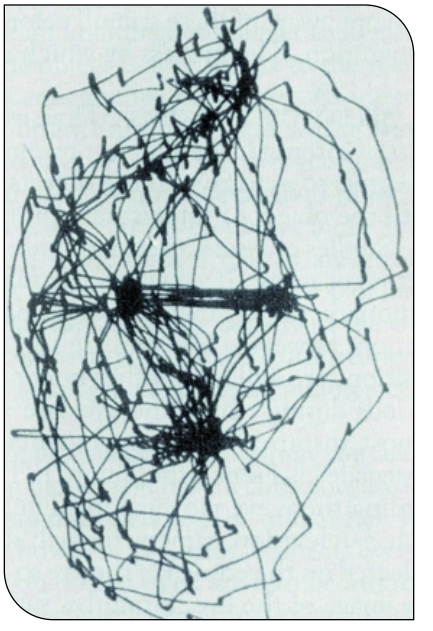


FIGURE 3. Eye tracking.

You build this level of visual detail by combining the information accumulated from tightly focused areas that were observed one at a time with the less precise, but far wider perception of everything within the field of view.

The Attention Span of an Adolescent Fruit Fly

Attention spans are naturally affected by a wide variety of variables. Logically, you'd expect a 16-year-old to lose interest in a lecture about the history of coal in northern Czechoslovakia more quickly than if he were listening to a stand-up routine by his favorite comedian. Visually, however, we're all pretty much in the same boat when it comes to attention spans. As I outlined with the preceding example of the prison, we construct a visual impression of the world around us through focusing on a sequence of small areas, where we observe each area for a very short period of time.

As far as the distribution of these brief spells of concentration goes, tracking the associated eye movements illustrates the areas of a scene that are important to the viewer. Figures 2 and 3 show a girl's face and the corresponding eye tracking showing the way a viewer might register her face.

Our eyes are rarely still for more than a few seconds, and it is our eyes' continual movement that builds up a detailed picture of our surroundings and lets us keep track of any changes, thus reducing the likelihood that we'll walk in front of a truck as we attempt to cross the road.

So, what use is any of this information to the videogame artist, who's already struggling under the burden of reproducing a two-headed skeletal dragon in fewer than 47 polygons?

Understanding the mechanisms of how we perceive the world can help an artist be more successful building the game environment, use resources more effectively, and create drama and atmosphere that enhances the gaming experience.

Giving Good Face

Perhaps the most obvious fact that arises from looking at the distribution of eye movements in Figure 3 is the amount of time we spend looking at the eyes and mouth. Naturally, when we look at someone, we concentrate on the features that communicate emotions most readily to us, and the eyes and mouth have the ability to transmit vast amounts of information very rapidly. Even from a photograph, we can infer a great deal about the subject.

The expressions in Figures 4 and 5 are perhaps slightly caricatured (exaggerating expressions in a game is often helpful), but it is still immediately obvious that the man pictured is angry or disgruntled,

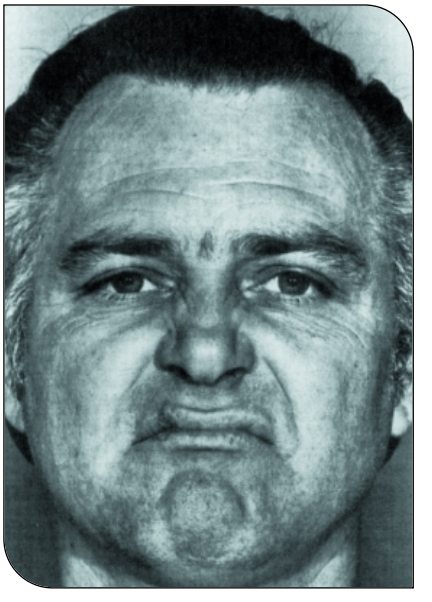


FIGURE 4. Grumpy man.

FIGURE 5. Sad lady.

whereas the woman is either sad or concerned.

The amount of time we spend creating convincing facial expressions in a game is often quite limited. Together with poorly executed textures that make characters look like their features have melted, many games overlook the importance of faces. Of course, exactly how important faces are depends entirely on the kind of game one is making. The more a character speaks, and the more time spent on screen in close-up, the more time a player will spend examining a character's face.

Purely in terms of polygon count, it is common sense to spend considerably more of the overall game budget for a character on the face than on exquisitely detailed boots, for example. One of the many characteristics that distinguish a high-quality animated film from its disposable made-for-TV counterpart is the effort that goes into conveying emotion through a character's expressions, rather than just relying on the script.

FIGURE 6. Too much information.

Game budgets and schedules being what they are, outside of the cinematics, many game artists content themselves with the kind of facial animation that makes Schwarzenegger's Terminator look like Jim Carrey when it comes to expression. While facial expression may have no direct bearing on gameplay, it affects the overall quality of the gaming experience as it relates in particular to story and characterization. Exactly how much time can be spent in this area is always a question of resources, however, and will depend on the individual project.

Feed Me with a Spoon

The world we create for our game, no matter how detailed and convincing, is going to be presented to the player through the medium of the screen. Virtual reality headsets have (mercifully for those of us with unruly hair) not yet staged a comeback of any significance, and the rectangle of our TV or monitor remains the display method of choice. This being the case, even though a player can adjust the viewpoint and examine the world as he or she sees fit, it is always viewed within this frame, and that is where focus begins once again to play a part.

While linearity in a game is often berated as the evil nemesis of quality gaming, no matter how complex a web of dynamic plot adaptation algorithms you employ, your characters will encounter significant locations in your game one at a time. It is in these distinct locations that games can often squander their carefully crafted visuals. A poorly executed setting can fail for a number of reasons, and it is worth remembering that it's not simply the visual impact that will suffer as a result. Game flow, plot development, creation of tension, and atmosphere are all equally at risk. Figure 6 shows an example of a scene with too much detail.

Take, for example, a game that drops players into an unfamiliar environment where a great deal of information (visual or otherwise) bombards them relentlessly as they wander around trying to discern what they should be doing. This problem comes down to bad design, but visuals within a game often compound the problem by either being unorganized or trying to create a richly detailed world that ends up being confusing because of this detail.

Most artists I know who work in games gaze at astounding pixel-feasts such as *Shrek* or *Final Fantasy: The Spirits Within* and fantasize about the not-too-distant future, when it will be games that are pumping out that kind of quality on the screen. No one knows when this dream will become a reality, but when it does, the fact will remain that a game has to accommodate a player, whereas a movie only has to satisfy a spectator. Players need to understand their locations far more than a spectator, as they have to navigate them successfully, completing tasks and/or fighting enemies on their way. A huge amount of detail can detract from gameplay if it is dispersed without any regard for focus.

Consider a small child learning to eat. Present him simultaneously with a bowl of mashed potatoes, a bowl of fruit puree, and a cup of milk. Now hand him a spoon and step back. What is going to happen? In all likelihood, the child will eat very small amounts of food and will have a great deal of fun distributing the rest around the room.

Now make the creakingly obvious metaphorical leap to the player, new to your game, who is presented with an unfamiliar world of characters, locations, dialogue, weapons, and so on, and is expected to negotiate them successfully. It's true that like a child, the player needs to learn through experimentation, with failure being an inevitable part of that experience. It's also true that the fun of splattering food around the room has a certain value associated with it. But if the goal is to eat, complete the meal, and be nourished, then supervised feeding, one spoon at a time, delivers results.

I'm not trying to make an argument for reducing games and their visuals to simple pedestrian journeys that require very little thought as the player is handed each stage on a plate. I am trying to suggest that each location within the game should facilitate exploration, plot development, and so on, through the careful presentation of information, visually or otherwise.

FIGURE 7. Heading toward darkness.

I Understand It, But Is It Pretty?

So, I've outlined some reasons for considering the way in which a scene is constructed with regard to its function within the game, but the idea of focus is also relevant on a purely aesthetic level. In this respect, the following two areas are worth considering: composition and lighting.

Composition

Composition is an area that can easily be overlooked. As a game is not static and the artist does not control the player's movements, the camera is out of his control in many cases. In film, the work of people like Ridley Scott illustrates how composition can be effectively employed in a moving scene; but still, these camera moves are constrained by the director, unlike in a game. Therefore, introducing the player to a new area through a cutscene or along a specifically designed route can be the best compromise available. In doing so, features that are important or attractive can be given prominence in the frame using a scene's best assets to maximum effect.

Lighting

Much has been written about lighting in games, but in this particular context it is worth noting that the use of light within a scene is perhaps the most effective way of focusing the player's attention. Whether it is simply the aesthetics of a particular location that benefit from lighting that picks out its most attractive features, or whether some more practical function is served (such as highlighting puzzle items or exits), a player's eye will be drawn to areas of light, in the same way that it will be drawn to areas of movement. With this in mind, contrast is a vital tool which not only allows the artist to control the focus, but to add drama. A bright light shining from within the next area can look inviting if the player's current location is dimly lit. Conversely, heading for an opening that leads into relative darkness can build apprehension quite effectively (see Figure 7).

The Payoff

Until we reach the point where the power of the platforms for which we're making games removes the technical restraints that hold us back as artists and designers, creating visuals for a game is always going to be about making the most of what we have available. Drawing the player's attention to certain areas of each environment and putting them together in such a way that they are both clear in their purpose, as well as visually impressive, can be tricky. The payoff, however, is a much smoother gaming experience, with less time wasted in frustrating and fruitless wandering around a level, as well as visuals that are focused on impressing the player with detail where it will be the most effective.

The 400 Project

At the 2001 Game Developers Conference, LucasArts' Hal Barwood gave a talk called "4 of the 400." He presented four rules of thumb for game designers, out of the (arbitrarily) 400 or more rules out there. As a friend and former co-designer of Hal's, I was intrigued enough to suggest we put together a project to try to glean more of the 400, and this new column represents the first time the results of that intention see print.

Our objective is to compile a list of practical rules that can be applied to help create better games, not just abstract observations of similarities among designs, or academic theories with no basis in the craft of game design. For each of these columns, we will include some kernel of design knowledge that you can implement right away and improve by some measure the design of your games.

The format for a rule is shown below, with a sample rule — "Provide clear short-term goals" — to get us started. Each rule will have five components, outlined further below. Until there is an existing database of these rules, it will be harder to list rules that it trumps, or is trumped by, but over time such cross-referencing will become easier. For this month, I've referenced other rules that should be pretty obvious to most game designers.

1. Each rule should be stated in a concise, imperative statement and paragraph:

Rule: Provide clear short-term goals. Always make it clear to players what their short-term objectives are. This can be done explicitly by telling them directly, or implicitly by leading them toward those goals through environmental cues. This avoids the frustration of uncertainty and gives players confidence that they are making forward progress.

2. The rule's domain of application. This includes both its hierarchy — for example "a rule about rules," "a rule about the development process," or just "a rule about games themselves" — and genre, for example, "applies only to RTS games or online games." This month's rule is a basic rule of game design and applies to all games directly.

3. Rules that it trumps (over which this rule takes precedence). It trumps the rule "emphasize exploration and discovery," because players should not have to discover their short-term goals. If discovery is warranted, it should be to discover the tools or information needed to achieve the clear, short-term goals, not to discover the goals themselves. It also trumps "provide an enticing long-term goal," as it is more important to have players know what to do next than to know simply that they have to kill the evil wizard/save the world/rescue the princess.

4. Rules that it is trumped by. It is trumped by the rule "make the first player action in a game painfully obvious." However, often that first obvious action in a game — read the paper, click on the wise old man, shoot the monster — should trigger an explanation of the first short-term goal beyond that.

5. An example or two from well-known published games where this rule is observed. When Hal Barwood and I designed INDIANA JONES AND THE FATE OF ATLANTIS, we gave the player explicit goals throughout the game by having the supporting characters guide the objectives. The initial theft of an artifact by a Nazi agent led the player (in the role of Indiana Jones) to Madam Sophia, who in turn presented Indy with his next objective, and so on. One short-term goal, such as "convince this character to give you an artifact," often triggered conversation with the character that led to the next goal, like "find the lost Dialogue of Plato."

Shigeru Miyamoto uses clear, short-term goals throughout all of his games. In SUPER MARIO 64 he uses explicit goals such as characters or signs that tell you how to move, jump, or swim, which are adjacent to appropriate obstacles. Other goals are implicit ones, as when you're left to explore the landscape at the beginning of the game with a large castle dominating the landscape and a drawbridge leading right to it. He also uses strings of floating coins to pick up as implicit goals that help lead the player into attempting jumps and using catapults or cannons pointing toward the coins.

More recently, Bungie's HALO does an admirable job of using the landscape itself and suggestions from both an AI companion and fellow Marines to channel players toward the next short-term goal.

NOAH FALSTEIN | *Noah is a 22-year veteran of the game industry. You can find a list of his credits and other information at www.theinspiracy.com. If you're an experienced game designer interested in contributing to The 400 Project, please e-mail Noah at noah@theinspiracy.com (include your game design background) for more information about how to submit rules.*

The Unimportance of Audio

Even if you're not your company's "audio guy," our new monthly game audio column will give you practical audio tips and ideas on how you can influence your games to have great soundtracks.

There are more important things in the world than sound for games. The smell of bread. Game design. Gas in the car. Surfing. Dating. War. Prayer. Fishing trips.

Often the worst audio happens when we forget how unimportant audio is. A sure sign that a band's performance is going to be awful, for instance, is if you happen to catch the band members talking about the upcoming show as "the most important gig of our lives." Run from that one. Don't even take the free tickets.

We have other sayings in the music business that reflect and amplify this wisdom. On *The Muppet Show*, George Burns once advised Rolf, the piano-playing dog, to "play like you're not getting paid." We often find it helpful to use the phrase, "It's only sound, it's not like there are any lives at stake."

However, I never trust a philosophy that isn't traveling with its equal opposite. "Haste makes waste" must be countered by "A stitch in time saves nine."

Therefore, I am compelled to state with equal conviction that there is nothing in the world more important than sound for games, if for no other reason than because it's my chosen line of work, and I'm certainly not in it for the money.

Perhaps this apparent contradiction is best reconciled by thinking that anything worth doing is worth doing right, but it's only by delving into the work that we slowly figure out what "right" is. As we spend more and more years learning to do something, we discover a basic truth: if we understand any one thing thoroughly, we can understand everything. I chose audio, you might have chosen software engineering or level design, somebody else might have chosen automobile racing.

What do we know after all those years of experience? We learned that considering audio either completely unimportant or completely all-important leads to massive screw-ups.

Problems that arise when we think audio is not important:

- Tools don't get developed to allow for efficient sound creation and integration.
- Audio doesn't get onto the production timeline.
- A friend-in-a-band gets hired to do the audio.
- There's no sound specifications for play-testing, therefore audio is not play-tested.
- Enough of this, you've heard it all a million times.

Problems that arise when we think too much of our audio:

- We get the highly unattractive "Most Important Gig of Our Lives" syndrome.
- We spend so much on licensed "famous-person music" that the budget only allows for a small quantity of music — which is then repeated and repeated and repeated and repeated.
- When a producer doesn't take chances, good musical "accidents" and innovation are impossible.
- Out of fear and theory rather than ears, we hire musicians and sound designers from the TV and film industry, who must then learn interactivity on the fly.
- Overblown audio budgets can (although I've only seen it once) break a project.
- Insane and elaborate interactive music-integration schemes can die half-ripe on the vine.
- Over-attention to audio can create very unbecoming situations. I once heard a good friend and a great producer brag about how he had thrown away two-thirds of the soundtrack to a game. Even if that two-thirds was substandard, I bet the player would have welcomed it after the 20th hour of gameplay.

By simultaneously holding the importance and the unimportance of audio in our minds, we can lean into the art of audio for games with every ounce of our earnestness, and not a bit of that attention will be wasted. We can take control, but still take chances. The audience hears it and feels it, we improve our skills and hone our tools, knock out our producers, blow away the investors, and perhaps even show our ideas and our work to thousands of players. We can raise the level of excellence of our industry, our community, and our craft. I feel like I got to do it a couple of times on titles such as LOOM, WING COMMANDER, and THE 7TH GUEST. I think Jeremy Soule did it on TOTAL ANNIHILATION, Michael Land did it on THE DIG, Peter McConnell did it on GRIM FANDANGO — there are a bunch more, and I hope that you've been on, or are going to be on, one of the teams that is responsible for one of these beautiful products, or better yet break new ground with a future team and project. It feels really good, and sometimes there's money in it.

NEXT MONTH: *Aaron Marks, author of* The Complete Guide to Game Audio, *talks about audio mastering for games.*

THE FAT MAN | *George Sanger is* Game Developer*'s audio advisor. Visit his one-of-a-kind web site at www.fatman.com.*

Distributing Object State for Networked Games Using Object Views

As game developers, we are continuously challenged to create richer and richer game worlds. Whether we are developing a 16-player multiplayer game, or a 10,000-player persistent world, making richer game worlds efficiently means we must be increasingly intelligent about how we distribute the ever-changing state of our game objects. This problem is further complicated by the diversity of the network connection characteristics of each player.

In this article, I'll describe a technique for managing the distribution of object state using an encapsulation mechanism called an object view. Object views provide a means for managing the distribution of object state on a per-object basis that is flexible and transparent to the game object. In order to describe what they are and how they are used, we'll also peer into the workings of a distributed-object system designed for multiplayer games.

Net Profit, Net Loss

As with many other areas of computing, some of the most significant problems inherent in distributing simulations have to do with resource management. In the case of networking, our primary concerns are with the limitations of the game clients and especially the nasty problem of controlling bandwidth utilization.

For most subscription-based massively multiplayer (MMP) games, bandwidth limitations are not based upon physical limits; rather they are based upon bandwidth costs. This means that proper bandwidth management translates into real dollars in a very big and measurable way.

Other techniques, such as those for masking lag and smoothing movement, are also essential for creating great multiplayer games. But for these to be effective, accountability must be had in the underlying implementation for the bandwidth limitation, whether constrained artificially or by the physical medium itself. After all, the bits have to actually arrive at their destination before they can do any good. Proper bandwidth management isn't just a networking problem, it's a whole-game problem.

But what does all this accountability have to do with object views? Before we get into the nuts and bolts of object views, let's talk a little about why we need them.

So Many Objects, So Little Time

At Monolith, we have been using object views as a fundamental construct in the development of our distributed-object system. A distributed-object system is a game system that manages the housekeeping chores related to the distribution of object state. It is the principal user of the relevant-set creation mechanisms, which in our implementation are provided by the world representation (see Figure 1). Relevant sets are collections of objects whose state changes need to be distributed immediately (if not sooner) if we are to ensure that a remote client's view of the simulation matches the actual state of the simulation. The topic of relevant-set generation is so large that it warrants its own separate discussion, so I won't be delving into it very much here.

In multiplayer games, we generally associate a connected player-client with a single, player-character-centered view of the simulation. Each client only needs to render a limited portion of the game world at any one point in time. Consequently, the state of all the game objects that are relevant to the rendered portion of the simulation must be up-to-date.

Direct data management vs. RPC. Distributed-object system implementations for both games and distributed simulations typically manage distribution of object state data rather than simply providing a general-purpose remote procedure call (RPC)-based mechanism. Why? The answer is rooted not only in our fundamental need to make the best possible use of the available bandwidth, but also, as we will see later, in our need to design a system that makes it as simple as possible for us to specify exactly how we want the component parts of our game objects to be distributed.

The responsibilities of a distributed-object system are conceptually quite simple:

1. Obtain a relevant set of objects for a client.
2. For each object in the relevant set, distribute any state that has changed since the last time that object was distributed to that client.
3. Repeat the preceding steps for each client.

As simple as the system seems conceptually, the devil really is in the details. Even if we are strictly using visibility-

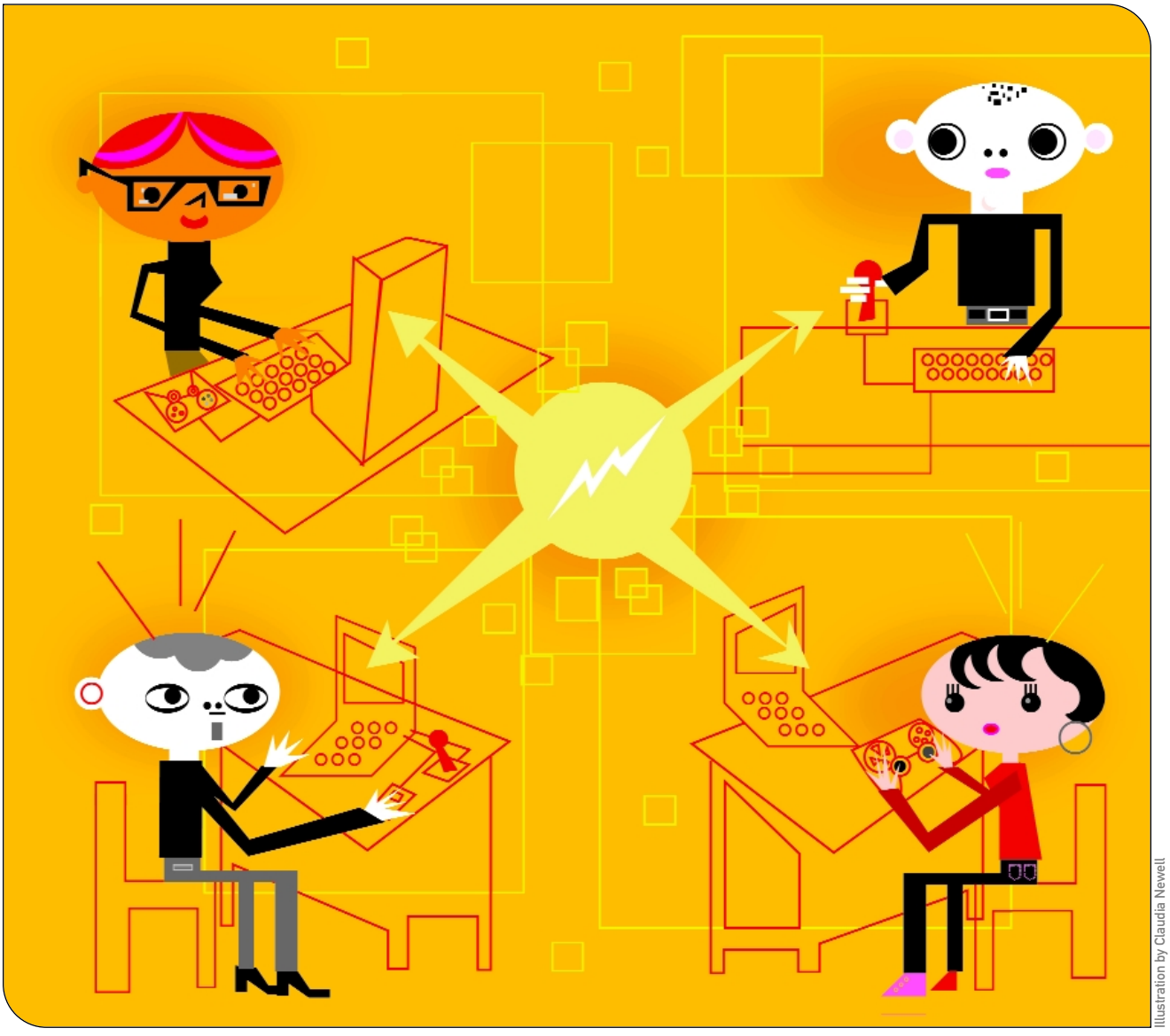


Illustration by Claudia Newell

based relevance determination, the full relevant set for a given client at any instant can be enormous. As an example, consider what happens when you direct your player-character to stroll up to the top of a nearby hill. As you crest the hill, the number of visible objects is likely to increase dramatically. Unfortunately, the amount of available bandwidth remains somewhat constant over time, so the distribution of objects in the relevant set must be managed carefully, using prioritization techniques that allow the most important state to be sent immediately and the less important state to be transmitted as soon as possible thereafter.

Multiplayer game objects. We've discussed that a key functionality of the distributed-object system is identifying, prioritizing, and selecting game objects from relevant sets, restricting the set of game objects to only those that need to be distributed. But only some of the components from a given object will need to be distributed. What are these components? To answer that, let's take a look at a simple game object.

Figure 2 shows the basic component parts of a simple game object that you might find in a generic multiplayer game. The object consists of three major groups of component items:

- **Visual and display-related items.** These are component items related to the visual state of the game object, including movement and position information. They very much need to be distributed. For player-character objects, this includes values that may only be displayed on a HUD (heads-up display) of the player controlling that character.
- **Game logic and AI-related items.** These are component items related to the game state of the object. In a purely server-based simulation, these items would seldom (if ever) be distributed to clients,

RICK LAMBRIGHT | *Rick is director of online technology for Monolith Productions in Kirkland, Wash., where he is leading an amazing team of developers working on a soon-to-be-announced massively multiplayer title. In his copious spare time, he enjoys composing ambient and trance music, and teaching Python to his twin sons.*

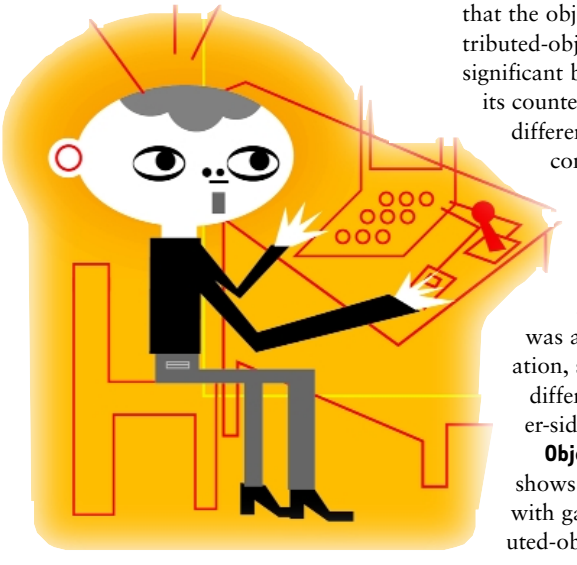
but could be distributed to a trusted entity, such as another server.

• **Housekeeping items.** These are component items, such as reference counts and pointers to internal structures. They are not distributed.

As our player-character roves around within the simulation, it will encounter new game objects, spend a little time hanging around near them, leave the area, and very likely reencounter many of the same game objects sometime later on. Since we only want to be sent updates for the items that have changed since the last time we encountered the object, something will have to remember the state that the object was in the last time we encountered it. To complicate matters, one client may have very different distribution requirements from another client for the same object. This is where object views come in.

Object Views

An object view is an instance of a custom class that knows how to access one or more components of a game object and track any changes to those components. Every object view is attached to a game object, and every object view also has a remote counterpart that is attached to a game object with a similar set of components. As changes occur to the states of the tracked components, the object view is responsible for communicating those changes to its remote counterpart. The counterpart is then responsible for applying those changes to the game object to which it is attached.



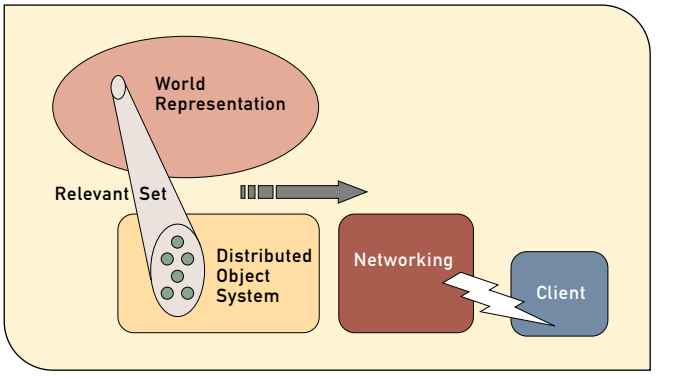


FIGURE 1. Major game components involved with object state distribution.

The distributed-object system itself is designed to interact with object views, not game objects. How the object view interacts with each game object is strictly a contract between the object view and the game object. The distributed-object system only distributes object views.

To access the game-object components (given a reference to a game object) efficiently at run time, each object view instance is created with full knowledge of which components of the game object it needs to track and how to access them. Hence, implicit in the nature of the object view is the notion of a binding to the game-object components that the object view will track.

The abstraction from the game object that the object view provides to the distributed-object system is one of its most significant benefits. An object view and its counterpart can each be bound to a different type of object and still communicate with each other for managing state distribution. This eliminates the requirement to use identical objects on both the client and the server. For us, this was an important design consideration, since our client-side objects differ significantly from their server-side counterparts.

Object view operations. Figure 3 shows how object views interact with game objects and the distributed-object system at a high level.

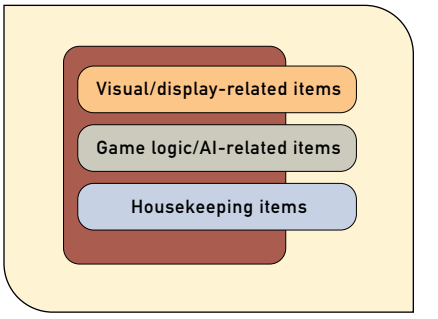
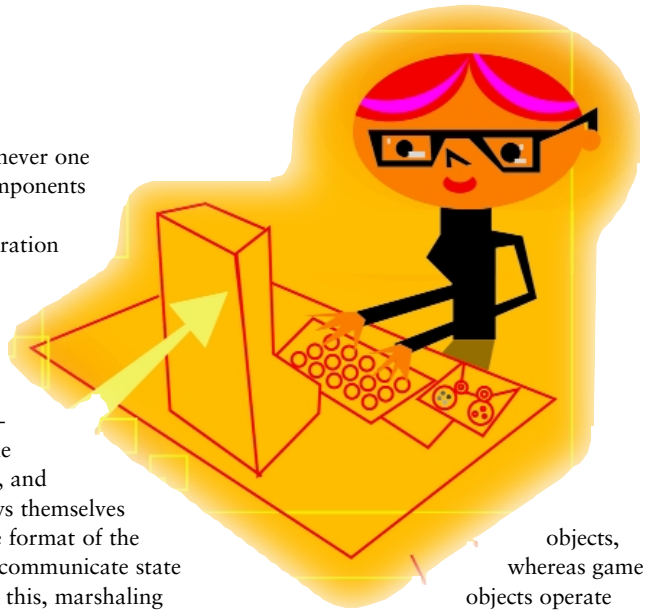


FIGURE 2. A simple game object.

Note that there is a one-to-many relationship of object views to game objects on the server, and a one-to-one relationship on the client. In client/server architectures, servers maintain connections to many clients, but the client typically has only one connection to a server.

The object view functions as a local proxy that remembers the state of each game object's distributed components from the last time it was distributed to a particular client. Since state distribution will only occur when game objects are relevant to a client, the state of each object view is potentially unique.

When an object enters the relevant set for a client, the distributed-object system first locates the client-specific object views for that game object, creating a new one if one does not already exist. Newly created object views on the server represent objects that will need to be created and fully initialized on the client before they can be rendered.



Either way, the process of determining exactly what state updates are needed and how the determination is made is strictly a contract between the object view and the game object. In order to ensure that the object view is granted the flexibility it needs, the distributed-object system requires every object view to provide two basic operations: pack-to and unpack-from.

The pack-to operation is called when the object view needs to be provided an opportunity to distribute its state. The object view determines whether or not any state updates are required, and is then responsible for marshaling those updates directly into the transmission buffer, packing them as tightly as possible in the process. Only the sending object view and its receiving counterpart on the other end of the connection can be trusted to understand the format of this data.

The object view's unpack-from operation is called up when state updates are received. This is typically a simple process of analyzing the received data and applying the updates to the appropriate components of the target game object. This also turns out to be a great time for an object view to provide event notifications to the game object — or to anywhere else in the game — whenever one or more specific components are updated.

A third basic operation that each object view should provide is solid diagnostics. Object view operations are deliberately mysterious to the rest of the system components, and only the object views themselves may understand the format of the data they utilize to communicate state updates. Because of this, marshaling errors will have downstream effects that can be difficult to debug without good diagnostics.

Tracking state changes. When it comes time to distribute the state of the game object, each object view will need to determine whether the components it is tracking have changed since the last time the pack-to operation was called. This requires the object view to remember something about the previous state of those components. There are a variety of techniques that the object view can utilize to track state changes; invasive techniques require special support from the game objects, whereas game objects operate obliviously to noninvasive techniques.

The determination of which tracked components have changed state will normally take place during the pack-to operation, and while the game object remains relevant for a client, the pack-to operation for its views will be called frequently. For this reason, the pack-to operation must be very efficient.

The most straightforward technique is for the object view to maintain its own copy of the game object components that it is tracking. If sufficient memory is available and the tracked items can be compared very efficiently, this noninvasive mechanism is hard to beat. Since the exact previous state of each variable is always available, the object view can be certain that it is only distributing state that differs on the target.

Adding a change counter to the game object is an evasive technique we have found particularly useful. We use this for complex objects that are tested frequently but whose state changes relatively infrequently. Each object view also has a change counter, and each time the state is distributed the view's counter is set to the current value of the game object's counter. By comparing the two counters, a very fast check can be made to see if any new changes have occurred. This technique could be used as an optimization for any object that is tracking more than a few items, but it does require that each game object be modified to ensure that its change counter is updated every time any

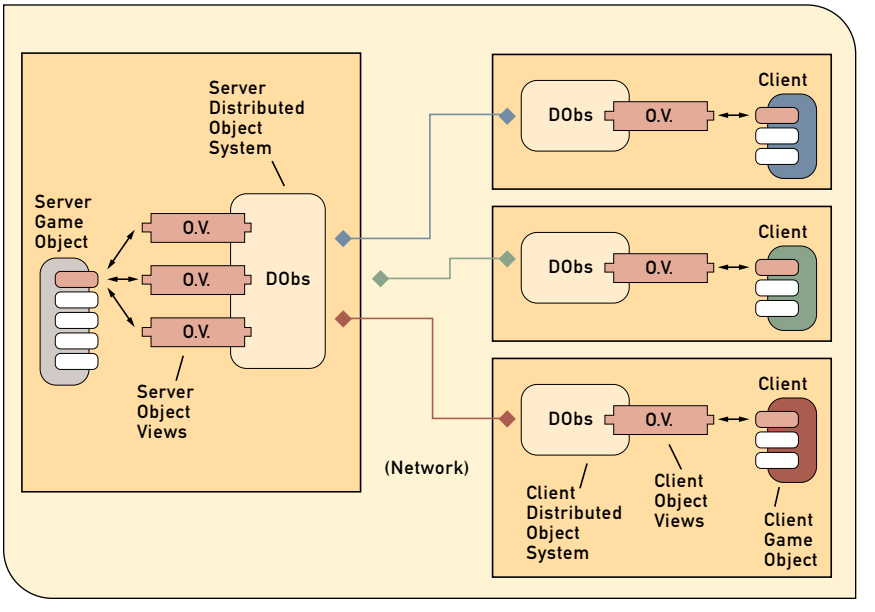


FIGURE 3. Object views in action (client/server).

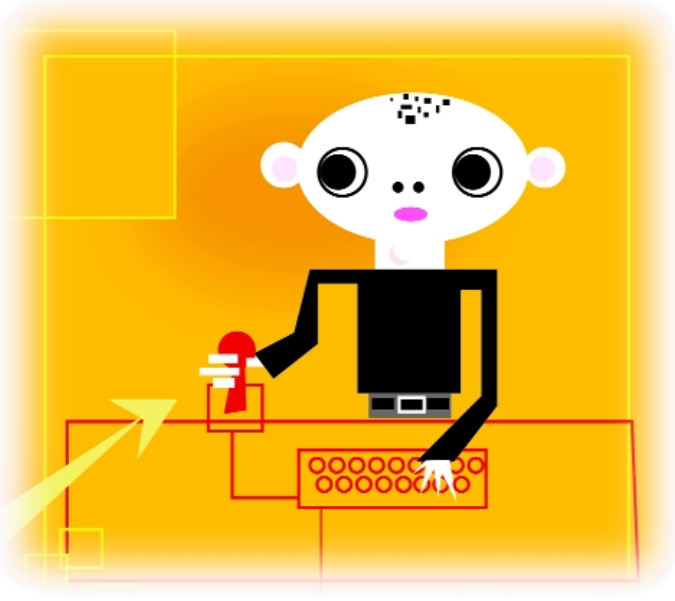
of the tracked components are updated.

Another invasive technique that we have seen utilized involves maintaining a bit set of change flags. This technique requires that the game object be designed to manage a bit set that is stored with the game object itself. Each bit in the set corresponds to a distributed component part. The object view keeps its own copy of the bit set and checks to see if its own copy matches that of the game object during the pack-to operation, in order to determine which component parts have changed.

Unfortunately, this technique suffers from three drawbacks. First, you must ensure that the corresponding bit is set every time a distributed component variable is updated. Second, if a component switches back and forth between a small set of states, then there is a significant chance that a value marked as changed would be sent to the target object even though it actually switched back to being in the same state as the target. This process wastes bandwidth. The third drawback is the most serious. The "changed" component bits on each object need to be cleared as soon as possible for optimal distribution, but they can only be safely cleared when state has been distributed to all clients. Because of that fact, this technique is really only practical for small-scale simulations where all clients need to be kept continuously up to date with the current state of all game objects.

Directionality. In client/server architectures we normally don't distribute object state from clients to servers for game objects other than the player-character object. Having multiple clients send competing updates to the same game object on a server might seem like a very strange thing to do in your game, but it might make complete sense for others, especially in peer-to-peer architectures. Though we tend to be quite security-paranoid when developing MMP games, there are no hard-and-fast rules. If sufficient safeguards are in place, any client could manage state updates and distribute those updates to a server or to other clients.

Complex objects. Game objects are typically hierarchical in nature. Any game-object component may itself be an object with its own component parts. This makes managing access to the items slightly more complicated than dealing with, for example, a set of items that are primitive types. You can generate or manually create custom object views for every game object and access each of the subcomponent items directly, one at a time. But if the same constituent objects are used as subcomponents in a wide variety of game objects, it is possible to implement your object views in a way that allows you to reuse a lot of code. To do this, you will need to create object views for the full range of component item types used by your game objects. This includes all object types and primitive types. Once this is done, complex objects can be managed by creating hierarchical object views that mirror the component hierarchy of the object.



Lifespan of an object view. Over the course of time, a player will potentially encounter tens of thousands of objects in a large simulation. A server would need to maintain all the object views permanently for every game object if it wanted to avoid the expense of re-creating them. This is memory-intensive not only for servers, but also potentially for clients as well. Fortunately, this problem can be handled fairly effectively using an active cache of object views. Old object views are then automatically purged from the cache over time if the game objects they track are not reencountered for extended periods.

Using Object Views

At the instant an object view needs to be created, a perfect opportunity exists to make some intelligent decisions. By checking the connection characteristics of the client, the distributed-object system can select an object view that is tailored for supporting specific clients. This also means that clients with unique communications requirements could conceivably coexist in the same game environment, sharing the game objects with clients that have completely different communications requirements. This could, for example, allow a client on a handheld device to share the game world with clients connected via a PC or game console.

Multiple object views, movement, and predictive contracts. The game objects in the preceding examples had only one object view bound to them. This was strictly for simplicity. In fact, the ability to divide the distribution responsibilities for an object's components into multiple object views is a powerful feature.

One use of multiple object views is for prioritizing game-object state related to movement. Because of its visual importance, movement-related object state is

usually distributed at a higher priority than any other object state.

When the distributed-object system creates or selects the object views for a game object, it utilizes a priority associated with each view to determine the initial order in which the object view set will be processed. By giving movement-related components their own object view and giving movement-related object views highest priority, movement-related information for all of the objects in the relevant set can be distributed first.

Object views are also natural places to handle prediction. In addition, managing movement prediction in the object view makes it possible to utilize different predictive contracts for clients with differing connection characteristics. For example, you could utilize a prediction technique for a client on a modem connection that was completely different from one with a broadband connection simply by selecting the appropriate type of object view when one needs to be created.

Name that tuning. Previously I mentioned that one of the reasons that we want our distributed-object system to manage distribution of our state data was because of our need to design a system that would let us easily specify how we want our game objects to be distributed. Applying distribution attributes to the data is necessary if we are to help tune how object state is distributed at run time. Tuning is a critical responsibility that is shared between the relevant-set mechanism and the distributed-object system.

To try to maintain a steady flow of traffic through the network, a measured allotment of bandwidth is calculated for each cycle. If a cycle exceeds its allotment, that affects the bandwidth allotment for the next cycle. When allotments are exceeded, the relevant-set mechanism must trim the set of objects to those that it determines are the most urgent to distribute. If the relevant-set mechanism undercompensates (that is, provides an excess of objects to distribute) for the available bandwidth on that cycle, the tuning support mechanisms of the distributed-object system and object views come into play. This also holds true when bandwidth is being underutilized. In this way, the two systems work together continuously to make optimal use of bandwidth.

The ability to tune how an object's state is distributed at run time is very important. By providing some specific information about how we want each game object to be distributed, we should be able to tune the system for optimal distribution. Here are some useful attributes that an object view can use for tuning how individual components, or groups of components, are distributed:

• **Priority.** A distribution priority can be set for each component item to designate which items are more important to distribute. When an object view is faced with needing to reduce the amount of bandwidth being consumed, it can select from the highest-priority items. As long as the object remains in the relevant set, lower-priority items will eventually be distributed during later cycles.

• **Reliability.** The ability to specify whether or not a component item's state should be distributed reliably (guaranteed) or if it can be distributed unreliably (not guaranteed), is a significant tuning option. When eligible for distribution, unreliable items will only need to be sent once. Delivery of the state update is never confirmed, so item state will not be present in case of a delivery failure. This attribute can have a great impact on overall bandwidth utilization in times of significant packet loss, but it must be used carefully. It is typically used for items that change very frequently and when a missed update has minimal impact. An object view could also choose to set the reliability attribute conditionally at run time.

• **Group.** Some component items will need to be distributed as a unit with others. The group attribute specifies that on a given cycle, unless all the member items of the group can be distributed, none should be distributed.

• **Direction.** For object views that support bidirectional state updates, the direction attribute can ensure that an object view only works in one direction. For example, the object view for a player object might need to be bidirectional when it connects to the client represented by that object, but unidirectional when distributing state belonging to a "foreign" player representing a different client. This can also be a security consideration on a server, preventing hacked clients from using bidirectional object views illegitimately.

• **Initialization only.** Components such as object IDs that will not change during the lifetime of the object can be tagged with the initialization-only attribute. After the initial distribution, these items will not need to be tracked by the object view, resulting in greater processing efficiency.

You should also provide a declarative means of assigning attributes to the distributed components of the game object. Ideally, this is part of the definition of the game object itself. A custom scripting language capable of defining game objects can build distribution-attribute assignments directly into the language itself. UnrealScript, for example, provides a `replication` statement, where delivery-related attributes can be specified for individual items of the class. In our own implementation, these attributes are assigned when the game object is defined

ACKNOWLEDGEMENTS

I would like to thank Ryan O'Rourke and Jeremy Friesen for their immense contributions in the subject area during the development of our own distributed-object systems, and for their assistance in reviewing this article.

using an internal compilation tool that generates source code for both the game object and its object views.

Where dragons dwell. Until a reliable transport protocol with predictable delivery is available over the Internet, simulations with time-critical delivery requirements will continue to use unreliable protocols, such as User Datagram Protocol (UDP).

Many complications can arise when object state is distributed using unreliable communications. Ideally, we want to use our limited bandwidth for transmitting only the most recent state of our game objects. Retransmission, due to packet loss, of old packets containing old state is a very poor way to solve the problem. Here too, object views have proved to be a very useful tool. In addition to their component-tracking responsibilities, they can also keep track of the success — or failure — of the delivery of state information to their remote counterparts.

FOR MORE INFORMATION

Frohnmayer, Mark, and Tim Gift. "The TRIBES Engine Networking Model." In *2000 Game Developers Conference Proceedings*. pp. 191–207.
www.gdconf.com/archives/proceedings/2000/frohnmayer.doc

Lambright, Rick. "Intelligent and Efficient Multi-Player Networking" (LithTech Development System 3.0 Whitepaper).
www.lithtech.com/general/pdf/IntelligentMult-Player.pdf

Lipkind, Ilya, Igor Pechtchanski, and Vijay Karamcheti. "Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations." In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '99). pp. 447–460.
www.cs.nyu.edu/vijayk/papers/view-oopsla99.pdf

LithTech Inc. *LithTech Development System Programming Guide*, Version 3.1. Kirkland, Wash.: LithTech Inc., 2001.

Sweeney, Tim. "Unreal Networking Architecture." Epic MegaGames, 1999.
http://unreal.epicgames.com/Network.htm

How do they do this? I'll leave the answer as an exercise for a rainy day.

The View, the Proud . . .

In this article, I've discussed how object views can be utilized as part of a distributed-object system to help encapsulate management of the distribution of object state. We also looked at how they can be used in the implementation of a distributed-object system.

At Monolith, we have found object views to be a very valuable tool in the implementation of our own distributed-object system. Object views have provided us with an extraordinary amount of flexibility, allowing us to create simple-yet-elegant solutions to a variety of the problems we needed to solve.

Integrating Your Game Engine into the Art Pipeline, Part 1: Integrating with 3DS Max

One of the art pipeline issues we've sought to resolve in developing AGE OF MYTHOLOGY is reducing or eliminating the time it takes for artists to see their content in-game. On previous games, the delay between an art revision's completion and its appearance in-game could range from hours to days. This delay meant that reviewing the in-game look for a particular art asset could take a long time. By creating a system where the artist can review in-game revisions in near real time, we've made the art creation and in-game review process significantly faster and simpler. In order to achieve this, we set about integrating our own game engine directly into Discreet's 3DS Max.

In the first of this two-part series, I'll discuss the issues and problems that have to be solved in order to integrate a Direct3D-based game engine into 3DS Max 4. I'll demonstrate how this is accomplished using a sample Direct3D model exporter and viewer running inside Max. In the second part of the series, I'll examine the same issues and problems in integrating a Direct3D-based game engine in Alias|Wavefront's Maya 4, and discuss the trade-offs between implementing both systems. Throughout this article, references to Max will be equally applicable to Maya, except for those cases where the discussion involves specific Max SDK programming.

Regardless of whether we're integrating with Max or Maya, there are three main issues that have to be resolved in order to integrate a game engine into the modeling package: getting the game engine, or at least the in-game renderer, to run in a viewport; determining how and when the game viewport should be updated with the in-game view of the scene; and obtaining keyboard, mouse, or other input to the game when running in the viewport.

The largest of these challenges is likely to be getting the game to the point of running as a plug-in for Max (or Maya) in the first place.

From Game to Game Plug-In

Even if your game engine runs flawlessly as a stand-alone executable, three main challenges await in order to make it able to be run inside a Max viewport (Figure 1). The largest of these tasks is converting the game engine into a Windows Dynamic Link Library (DLL) so that it can then be converted into a Max plug-in.

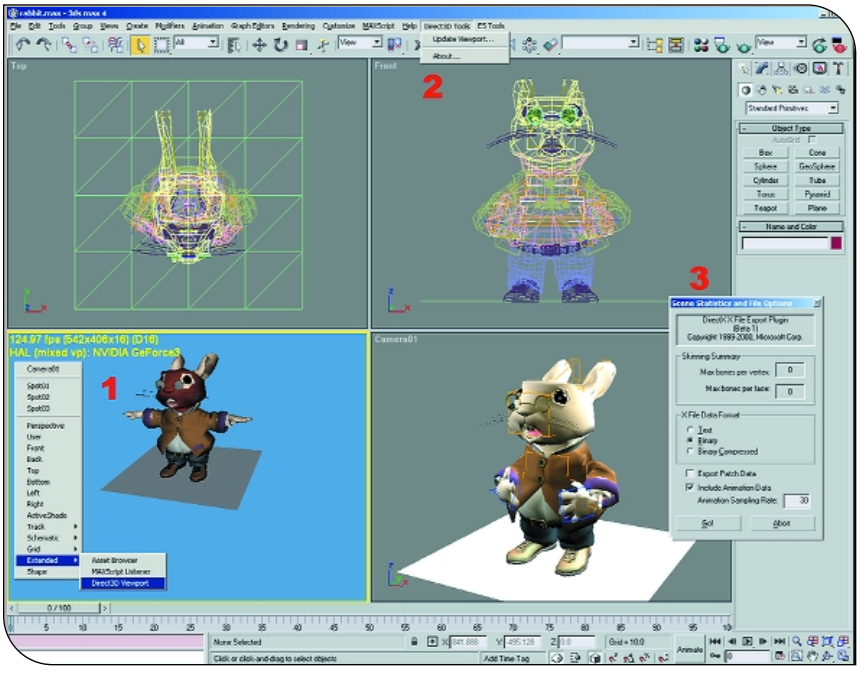
Making the conversion of the game engine a two-step process — first into a DLL, and then into a full-fledged plug-in — significantly simplifies the debugging process. This allows you to make sure the game functions properly as a DLL first, and, with that completed, worry about getting the game to run in the viewport within Max.

The actual conversion of the game engine into a DLL requires changing various compiler options as well as replacing the `WinMain` function. In an executable, `WinMain` typically contains a message loop that drives the processing of Windows messages and the game itself. Now that the game is running as a DLL, a separate thread will be needed to drive that processing. The accompanying source code project, SkinnedMeshDLL (available from the *Game Developer* web site at www.gdmag.com), is a version of the Microsoft Direct3D sample .X file viewer program SkinnedMesh, converted from a stand-alone executable into a DLL (Figure 2).

The new DLL version of SkinnedMesh exposes two functions: `startGame` and `stopGame`. These start and stop the running of the SkinnedMesh sample program, respectively. `startGame` calls the helper function `createRenderWindow`, which initializes the Direct3D framework and creates a thread (`ThreadProc` function) that will drive the framework's update function. `stopGame` simply posts a `WM_CLOSE` message to the Direct3D framework to tell it to shut down. The DLL can be run using the included unit test program unitTest.exe (Figure 2).

Now that the game is being driven by two threads — the one from the calling application and the DLL's own update thread — care must be taken to make sure that the game doesn't try to shut

HERB MARSELAS | *Herb is a game programmer at Microsoft's Ensemble Studios. He is currently working on AGE OF MYTHOLOGY. In his spare time he's building a molecular-sorting interocitor, and wondering how much Xbox really does weigh. He can be reached at herbm@microsoft.com all hours of the day and night.*



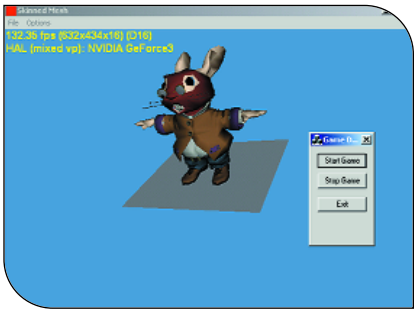


FIGURE 1 (left). Composite screenshot showing the three main interface components of the sample integrated Direct3D viewer running in Max. (1) Direct3D viewport itself, and its location in the extended viewport menu. (2) Tools menu with Direct3D viewer and game options. And (3) export options that are used to display the model in the Direct3D viewport.

FIGURE 2 (above). The Direct3D .X file viewer running as a DLL can be started and shut down repeatedly from the unit test application.

down while still rendering, or resize the rendering window in the middle of an update. To guard against this, critical sections have been added around the renderer update, application shutdown, and window resizing so that the application doesn't inadvertently try to destroy a Direct3D resource that's in use or use one that's already been destroyed.

For safety's sake, the Direct3D device creation has also been changed to inform Direct3D that the application is running in multiple threads. This flag can be removed if the application is vigilant about managing its own critical sections, but it's easy to forget and is added in for completeness of the sample code. In practice, most games will need to handle their own multi-threaded state with critical sections anyway. In this case it should be relatively easy to handle critical sections around Direct3D API calls as well. Handling this on the application side will remove the need for the Direct3D threading flag to be passed during device creation, and can result in some speed increase, as Direct3D won't have to manage its own critical sections. It should be noted, however, that this can cause serious messages from the debug version of Direct3D relating to the fact that API calls are being made from multiple threads, even though the multi-threading flag wasn't passed. In this case, this is perfectly fine and can be ignored.

The second issue that must be addressed is global variables. When the game ran as an executable, the global variables were initialized when the executable was loaded, the game ran, and then the executable was unloaded when the game ended.

Now that the game is converted to a DLL, the game can be run multiple times after being loaded. However, the global variables are only initialized to default values during the initial loading. Because the global variables are in an unexpected state, the game's shutdown code will need to put global variables back into a known state when the game shuts down so that the next start-up of the game doesn't crash.

The final problem is to make sure that the main game window can run as a child window. As most games run their main windows as top-level, parentless windows, window-creation flags, mouse positioning, and input code may need to be changed to take into account the fact that the game window will be a child window inside Max.

Once the game itself is running as a DLL, the next step is to make the conversion into a Max plug-in that can run in a viewport.

Getting the Game In

Converting the game DLL into a Max plug-in is a relatively straightforward process of changing the file extension from .DLL to a recognized plug-in type from Max (such as .GUP) and adding a few well-known functions that allow Max to load the plug-in correctly. What kind of plug-in does this need to be? That is the main question.

While Max 4 does add some support for Direct3D 8, the support is only available in a few very specific places (Figure 3). Direct3D pixel shaders are only supported as textures in the material system (`ID3D8PixelShader`), and a single vertex shader is available in the modifier stack (`ID3D8VertexShader`). Unfortunately, neither of these will help us render a full object, let alone an entire scene. There's also the

Direct3D graphics window (`ID3DGraphicsWindow`) that Max uses for rendering its ActiveShade viewport, but it doesn't expose any functionality that will allow us to use it from our own plug-in.

Because Max doesn't provide the access required to use its own Direct3D device, the game viewport will have to manage its own Direct3D device and window. This can be done by creating an extended viewport (Figure 4). This method will also simplify the Max integration because there will be no need for Max SDK calls to be placed in the game engine itself. All the Max integration can exist as a very thin layer that loads and starts the game DLL.

Max plug-ins that function as extended viewports fall into the category of General Utility Plug-ins (GUPs). These are plug-ins with the file extension .GUP and are loaded by Max at start-up. Once loaded, the GUP will create and register the extended viewport with the Max viewport system, allowing it to be displayed in the Extended Viewports list (Figure 1, #1). Another benefit of GUPs is that they have virtual free rein over Max and can access just about any part of it. This property will help later when we need to integrate into the Max menu system (Figure 1, #2).

Converting the working game DLL into a Max plug-in requires adding four exported functions which allow Max to verify the version of Max the plug-in supports and to query the plug-in for the types of interfaces supported (Listing 1). The interface mechanism is similar in some ways to the Microsoft Component Object Model (COM). There are some key differences, however. Whereas COM queries for the type of supported interfaces and instantiates interfaces as needed, Max queries for the number of supported plug-in interfaces (`LibNumberClasses`) and requests instantiation of a class-factory for each interface in turn (`LibClassDesc`). Max then uses the returned class factory to instantiate instances of the actual plug-in.

The accompanying sample code contains a project called SkinnedMesh, the next evolution of the SkinnedMesh DLL. This is the SkinnedMesh sample code converted completely into a Max GUP. It can be run by copying the d3dviewport.gup into the plug-ins\ subdirectory in the Max folder and restarting Max. The viewport is available as "Direct3D viewport" under the Extended Viewports list available from the Max viewport right-click context window (Figure 1, #1).

The GUP start-up function `SkeletonGUP::Start` (Listing 2) instantiates and registers the extended viewport window used by the game. The game itself isn't started until the user selects the Direct3D viewport in the Extended Viewports list (Figure 1, #1), and Max then calls the `SkeletonViewWindow::CreateViewWindow` function (Listing 3) which creates and runs the game.

When the user changes the viewport from the Direct3D viewport to another type of viewport, Max calls the `SkeletonViewWindow::DestroyViewWindow` function (Listing 3). However, instead of destroying the game window and all the Direct3D resources, it is paused until the user next invokes the viewport.

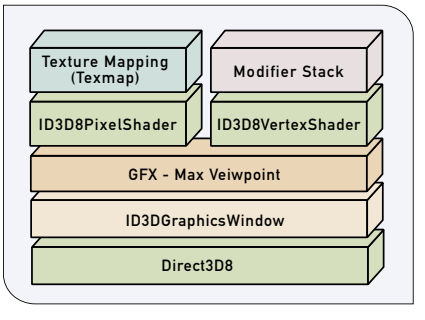


FIGURE 3. Direct3D8 functionality is exposed via the material texture mapping system, the modifier stack, and the graphics window.

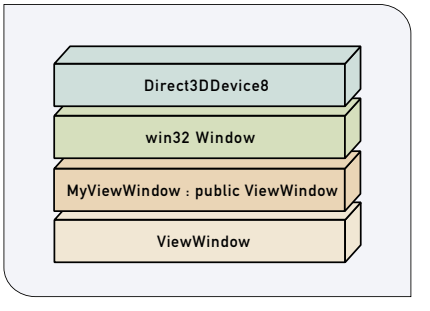


FIGURE 4. The game DLL manages its own window and Direct3D device as an extended viewport plug-in.

With the game running a viewport, the next step is to get data from the Max scene so it can be displayed.

Interacting with Max

Updating the viewport can be done in several ways. The simplest way would be for the viewport to continually look for Max scene changes and update them in real time. Unfortunately, it's probably too slow to parse the Max scene graph continually for new information.

An alternative is to update the viewport only when it receives focus. This can be fraught with problems, though, as users may not want to reparse the Max scene every time they want to move the camera around in the in-game view. Additionally, due to problems with the Max input system, the viewport window won't be assured of getting input focus messages (`WM_SETFOCUS`), causing additional problems. This could be handled by looking for mouse messages (such as `WM_LBUTTONDOWN`), but then you're back to the problem of determining when users want to update the in-game viewport as opposed to just changing the in-game camera position. The best alternative is to provide a user interface in a menu or dialog box that the user can use to alert the in-game view to update itself with data from the current Max scene.

In the accompanying sample code, the user is provided with a new menu, Direct3D Tools (Figure 1, #2). One of the options in this menu is to update the in-game viewport. When selected, users are presented with a dialog box (Figure 1, #3), prompting them for what they want to display in the viewport. Based on the Direct3D Max .X file exporter, this allows the user to export a single frame or a complete animation into the in-game viewport. This exporter Max plug-in previously was its own plug-in, but for use here it has been compiled directly into the viewport plug-in. This simplifies the management of the plug-ins, because we can call its export functionality directly without having to load it separately or otherwise search for it.

By reusing the Direct3D .X file exporter, we write the scene data to a file and then tell the game viewport to read it back in. This provides a clean break between Max and the game code, and as most games already have file-loader code lying around, there's no reason not to use it.

Alternatively, the game viewport could operate directly on the Max data structures or just use in-memory data structures to transfer the memory from the exporter to the game. Either of these methods can work just as well, but they do require more new code to be written, and they increase the amount of Max code that can end up in the game code.

Although the standard and prescribed Max SDK method for creating menus is to use the complicated parameter block system (`PARAMBLOCK2`), the menu that presents the user with the option of updating the viewport is a standard Windows menu.

While using `PARAMBLOCK2` might be useful in some applications, it's overkill for the simple level of menu integration we require; another alternative would be to use MaxScript in the menu and call into functions exported by another plug-in. However, by just using Win32, all the menu code takes no more than a few lines to create and handle in a single centralized location.

The menu code is inserted during the start up of the GUP (Listing 2). The main Max window is subclassed so that we can handle the messages for the menu entries we'll create, and then the `SkeletonGUP::addnewMenus` function creates the menu and inserts it into the Max main menu.

In creating these new menu entries, the Win32 function `SetMenuInfo` is used to associate a special tag value with each menu entry. This is needed so that we can handle our own menu entries during the Windows `WM_INITMENU` and `WM_MENUSELECT` messages. If the menu entries we added were passed along to Max, it would throw an assertion because it didn't recognize the menu identifiers.

So now the user can select data from Max and view it in the game viewport. Interacting with the viewport itself remains the final problem.

Getting Your Input

As mentioned previously, Max has problems getting all the Windows input messages to an extended viewport, and random messages such as "focus" (`WM_SETFOCUS`) seem to disappear into the abyss. Complicating this is the fact that Max has its own set of accelerator shortcut keys that need to be enabled (`EnableAccelerators`) or disabled (`DisableAccelerators`) as input focus changes between Max and the game viewport.

This message loss can be fixed to some degree by subclassing the parent window of the game viewport and triaging its messages to help you figure out who has, or should have, input focus. However, a better alternative is to use DirectInput for mouse and keyboard input. This is less of a hack, but it still has some issues, as it's possible to continue to collide with Max's accelerator keys. Another method altogether is to create a mode that locks the input focus to the game window and provides a more modal change between being "in-game" and "in Max."

LISTING 1. Plug-in registration functions required by Max.

```
__declspec( dllexport ) const TCHAR*LibDescription(void)

__declspec( dllexport ) int LibNumberClasses(void)

__declspec( dllexport ) ClassDesc*LibClassDesc(int i)

--declspec( dllexport ) ULONG LibVersion(void)
```

LISTING 2. GUP plug-in start-up code registers the new extended viewport window so that it's available to the user, and subclasses the main Max window to handle menu and other messages.

```
DWORD SkeletonGUP::Start()
{
   // create and register extended D3D viewport window

   SkeletonViewWindow*skvw = newSkeletonViewWindow(this);

   Max()->RegisterViewWindow(skvw);

   // store off MAX root node

   gpRootNode = Max()->GetRootNode();

   gpcontext = this;

   // setup window subclassing

   HWND hwnd = MaxWnd();

   gpMainWndProc = (WNDPROC) GetWindowLong(hwnd, GWL_WNDPROC);

   SetWindowLong(hwnd, GWL_WNDPROC, (long) SubclassWndProc);

   addnewMenus(hwnd, GetMenu(hwnd));

   return GUPRESULT_KEEP;
}
```

Each of these systems has its trade-offs, and the choice of which system is best will depend on the amount of viewport interactivity that's required and to what extent the in-game viewport needs to act like just another Max viewport. In the case where the artist simply needs to view a model in the in-game viewport to make sure that the visual quality is acceptable, using the Win32 input system and enabling and disabling Max accelerator keys is probably acceptable. On the other end of the spectrum, if the in-game viewport is being used as a large-scale level editor, with the artist or designer placing and editing assets in-game in the viewport, then the modal input system is a better choice. In the end, there are a range of options and combinations that could be used.

LISTING 3. To save time, the viewport creation function only instantiates the in-game viewport once. The in-game viewport is hidden in response to being destroyed, and reshown if it already exists.

```
HWND SkeletonViewWindow::CreateViewWindow(HWND hParent, int x, int y, int w, int h)
{
   if (IsWindow(g_d3dApp.GetMainWindow()))
   {
      pauseRenderer(false);

      return g_d3dApp.GetMainWindow();
   }

   return createRenderWindow(ghInstance, hParent, x, y, w, h);
}

void SkeletonViewWindow::DestroyViewWindow(HWND hWnd)
{
   pauseRenderer(true);
}
```

What about Max 3.x and gMax?

Max 3.x and gMax users can use these same basic techniques, and code only requires a recompile to run under either program. This is because the level of integration is very lightweight, and more code sits on top of the Max framework rather than integrating tightly into it.

Max 3.x will present more challenges, however. Even though the code will work without change, the Max 3.x input system has even more problems than Max 4 does. Just like Max 4, this can be somewhat solved by use of DirectInput and/or judicious use of subclassing of the windows in the Max 3.x viewport hierarchy. However, this may be more effort than one wants to try to handle. A modeless control window is an alternative, as is the use of an input device, such as a joystick, that can be read via DirectInput and won't interfere with the Max 3.x input system.

Common Problems

For development of Max plug-ins, the retail version of Max is pretty unforgiving when it comes to handling exceptions and crashes; this can make debugging a plug-in a trial-and-error process involving copious numbers of trace statements. The "debug version" of Max offers a solution to this problem.

Unlike the retail version of Max, the debug version includes symbol information and some source code that enables the debugger to provide a callstack that significantly speeds the debugging process. The debug version of Max is available through the Discreet Sparks developer program. Find out more information at their web site: http://sparks.discreet.com.

When you are actually running a Direct3D-based game viewport, the game often can't successfully create a Direct3D device, or it sometimes runs at only a few frames per second when it should run many times faster. Assuming that the installed graphics card supports Direct3D and the driver is up-to-date, the cause often turns out to be a Max configuration issue. While some graphics drivers can run Direct3D and OpenGL or HEIDI applications at the same time, some drivers just can't. Setting the Max Display Driver to Direct3D in the Viewports tab under Customize > Preferences will solve the problem.

A somewhat related problem is that a Direct3D-resource-hungry game can find itself aggressively competing with Max for driver and video card resources. This can result in slowdowns, strange Max error messages, or even crashes. If this situation occurs, there's little that can be done except to scale back the game's own Direct3D resource usage.

Caveat Developer

Integrating a Direct3D-based game engine into 3DS Max 4 is not an insurmountable problem; there are two main issues that have to be overcome in order to achieve this, however. While Max 4 touts Direct3D 8 support, the support is not to the level to be generally useful for development. Also, due to the way that windows input and focus messages are handled, the Max input system could potentially cause a lot of problems.

Next month, I'll examine these same issues in integrating with Alias|Wavefront's Maya 4, and discuss the trade-offs between implementing both systems.

FOR MORE INFORMATION

Discreet 3DS Max Developer Program
http://sparks.discreet.com

Microsoft Direct3D .X File Exporter for 3DS Max
http://msdn.microsoft.com/directx, in the downloads section under DirectX 8.1 SDK Extras.

Game Developers Conference 2002 Preview

Every March, game developers from around the world congregate in San Jose, Calif., for the annual Game Developers Conference (organized by CMP Media, publishers of *Game Developer*). In order to give readers a better idea of what to expect from this year's conference, we caught up with several different speakers from different tracks to talk about their plans for their sessions. This year's conference takes place March 19–23, with more information available at www.gdconf.com.

Marc LeBlanc

Lead Designer, Visual Concepts
Game Design Tutorial: "Game Tuning Workshop"

What's unique about your workshop that will make it a valuable two-day investment for game designers? The workshop is unlike anything else at the GDC. Participants will spend most of their time working in small groups, playing games, analyzing games, and solving game design problems. It's a unique opportunity to stretch your game design muscles and work with a lot of other smart people in the field. We've got a very experienced and talented faculty from all over the game industry, and a relatively small student-to-teacher ratio, about 15 to 1 (for this reason, I encourage those interested in the workshop to preregister for it, as space is limited). And by the end, the students have usually taught the faculty a thing or two as well.

How do you begin to enumerate various kinds of fun into aesthetic models for group-discussion purposes, when people tend to have varying opinions of what constitutes fun? The whole point of the aesthetic-models approach we use is to acknowledge the many kinds of fun that are out there; not only do different people like different games, but different people are going to like different things about the same game. If you start your game design with a formal understanding of the kinds of entertainment you're trying to provide, you have a much better chance of hitting those nails on the head.

Are the techniques you present in your workshop applicable to any game genre? Absolutely. The workshop focuses on game design models and techniques that transcend genres and media. We try to hit on a number of different genres in our exercises, and sometimes we use nondigital games, such as board and card games, to convey ideas that exist in digital games as well.

Is the workshop just for game designers? Game designers are definitely our primary audience, but we welcome folks from all disciplines. On many projects, design responsibilities are often shared by designers, producers, programmers, and artists, and people in all of these roles can benefit from the workshop.

How many years have you spoken at GDC? Why do you choose to do so? This is my fourth year on the GDC faculty. I enjoy speaking, and I feel like I have something to contribute. In this adolescent stage of the history of games, I think it's important for people to contribute what they can.

Rob Fermier

Lead Programmer, Ensemble Studios
Programming Lecture: "Building a Data-Driven Game Engine: A Case Study from Age of Mythology"

What will your session cover? The session covers our experiences trying to make a very strongly data driven RTS engine for Age of Mythology. It covers the high-level data architecture of several engine systems, such as UI and database systems, and some real-world analysis of how they worked, and more importantly, how they didn't work.

How does the data-driven system devised for Age of Mythology build on, or completely depart from, the previous Age games? The previous Age games had a strong data-driven component, so Age of Mythology's engine definitely builds on a lot of architectural principles from our earlier games. Age of Mythology takes it significantly further, extending those notions to almost every part of the game engine. Because Age of Mythology has been rewritten from scratch, we were able to be much more aggressive about being heavily data driven — though of course, it meant we had a lot more basic ground to cover as well.

What were some of the challenges you faced implementing a data-driven engine that you'll be sharing in your presentation? In addition to the obvious technical and schedule challenges, some of the more interesting problems we ran into were issues such as: How do you handle a steep learning curve for exposing complex tools to nontechnical creative staff? When data-driven systems become highly complex, how do you determine who is responsible for fixing the bugs? What are some of the security and hacking implications of exposing powerful data-driven systems to the end user? And I'll look at how leveraging existing technologies like XML and SQL can be a big benefit, and also how to grapple with their limitations.

How many years have you spoken at GDC? Why do you choose to do so? This is my first year speaking at a full GDC, though I did speak at a GDC Road Trip a few years back. Participating in GDC as a speaker is a great opportunity to share

development experiences with the rest of the industry. Fostering good communication between all the development houses in the industry can only help us all make better games in the long run, and that's what it's really all about.

Ian Baverstock

Business Development Director, Kuju Entertainment
Business & Legal Lecture: "Mobile Phone Games: Where Are They Now and Where Are They Going?"

What will your session cover? It will cover the current state of the mobile game market and some speculation on where I think the market is going. This will be particularly from a developer's point of view.

After a year or so of serious attention being paid to mobile gaming, do you see the hype and reality getting closer or farther apart? The reality is definitely getting closer. I think that mobile gaming has always been well served by realistic developers, as they didn't have the dot-com money to simply burn cash in the pursuit of some nebulous future market. I think the operators and other distribution channels are starting to focus on reality, and I think that real consumer markets are starting to shake down the business models that we'll use for the next few years. Having said that, the shakedown process is definitely not over yet.

What do independent developers need to know about approaching the mobile market from a business standpoint? The main thing is that it is very different from the retail market — different consumers, different distribution models, and a very different price point.

How many times have you spoken at GDC? Why do you do so? This is my first talk at GDC, although I did talk at GDC Europe last year. It's a fantastic opportunity to share ideas with some of the smartest people in the business.

Albert Chen

Senior Level Designer, Factor 5
Level Design Lecture: "May Time Be with You: Level Designing Rogue Leader" (with Chris Klie)

What will your session cover? The session will cover how the Rogue Leader level design team managed to deliver levels in nine months without sacrificing quality or fun. We will highlight the changes we made in our approaches to the design process, level content creation, scripting, and team organization.

What facet of level design remains the most neglected by level designers? I think that some level designers spend way too much time focused on story or other non-gameplay-related elements in their levels, and not enough time developing, play-testing, and refining gameplay. Yes, things such as story and cutscenes are important to a certain extent, but I believe some of us have forgotten what we're in the game industry to do. If you want to make movies, be a porn star.

How did the feedback cycle work for level designers on such a short project as Rogue Leader? We did peer play-testing among level designers and invited people in and out of the company to try the game. The LucasArts testing crew also gave us very important feedback and were fantastic with cool suggestions and ideas. If you want feedback from hardcore gamers, you would do well to talk to testers.

How many years have you spoken at GDC? Why do you choose to do so? This is the first time I'll be speaking at GDC. I feel that because of the tight development schedule, we were able to discover and rediscover ways to make a level designer's life a little easier. If someone takes something away from our session and manages to go home on time or spend less time in crunch mode, this talk will have been worth it.

Tetsuya Mizuguchi

President and CEO, United Game Artists (a member of the Sega Group)
Game Design Lecture: "Rez: The Synesthesia That Games Invite"

What will your session cover? My session will focus on the concept of synesthesia — the use of one stimulus to invoke another. For example, in Rez, the musical shooter, the music affects the game and graphics, and the game and graphics affect the music as you play. My session will also explore the discoveries made while translating new concepts and ideas into groundbreaking games.

Tell us a bit about this concept of synesthesia. What inspired it? The concept of synesthesia is the melding of the senses, such as hearing sight or feeling sounds. This concept and the art of Wassily Kandinsky inspired me. I was motivated to create Rez after a party I went to in Europe in 1995. The sounds and the light were all synchronized. I felt something. I imagined orchestrating the sounds, lights, and music. I thought that this would be fun for a gamer.

How else would you like to see the innovative use of audio expand into game design? There are many ways to incorporate audio into games. One way to expand the audio experience is through peripherals. For example, the transvibrator used with Rez allows the player to feel sound through the peripheral in addition to hearing it as they progress

through the game.

How many years have you spoken at GDC? Why do you choose to do so? I have spoken at GDC twice. I choose to speak at GDC because it is an event especially for game developers. It is an opportunity to share my experience and discoveries with other professionals in my field.

Jason Della Rocca

Program Director, International Game Developers Association
Sessions: Numerous IGDA-sponsored sessions and events

[Editor's note: The IGDA is an independent, nonprofit organization under management contract with CMP Media, publishers of ` and organizers of the Game Developers Conference.] **What are the IGDA's main goals for the conference this year?** We really want to further demonstrate to the development community that the IGDA is here for them and we are helping to make a difference and effect change within the industry. More importantly, we want developers to understand that the IGDA is their association and they need to leverage it and work together to foster the industry. In part, the IGDA is a means to actually do something once you leave the conference. So many events I go to are great, but once I leave it's all over. In our case, you can attend a panel on software patents, for example, and then turn around and join the IGDA's committee dealing with this issue, and actually contribute to making a difference on an ongoing basis.

Tell me a bit about the two-day Academic Summit. Warren Spector and Doug Church, along with other Education Committee members, have been doing a lot of great work with the academic world. The Summit is an extension of this work and will focus on research relations and game development and design curricula. On the research front, a greater level of collaboration would benefit both sides, but progress is often marred by a lack of respect and understanding; we will work to build bridges. On the curriculum front, we are seeing a growing demand for programs that teach the making and discussion of games. Having the industry drive curriculum discussions is only natural. I am very excited for this summit and expect great things to come of it — there are a lot of great developers and pioneering academics involved.

How is the IGDA building on the Game Developers Choice Awards, which debuted at the show last year? We are doing our best to maintain their level of sincerity and not get caught up going the glitz-glam route. We're trying to build on the credibility, respect, and honor of the awards.

What's the hot-button issue for game developers this year? There are lot of things, many of which will be covered in the IGDA's track and other GDC sessions, but I'd say there is an overwhelming sense that developers are real underdogs against publishers. Whether hype, truth, or some combination thereof, there is a perception of an uneven balance of power. Several sessions are looking into balancing things out.

Dave Ranyard

Senior Programmer, Team Soho, Sony Computer Entertainment Europe
Visual Arts Lecture: "UI Case Study: This Is Football 2002" (with Andrew Hamilton)

What will your session cover? It covers the development of the interface for This Is Football — World Tour Soccer in the U.S. — or, "the dirtiest job in videogames."

The interface for this game is big. And I mean really big. Hundreds of screens displaying all kinds of fascinating sporting detail. And not only that, it comes out in loads of different languages and has different SKUs for different licenses. Development of such a monster is a pain in the, well, you-know-where. To alleviate some of this, we built a system that allows graphic designers to change the look and feel of the interface without programmer intervention. We built our own Flash renderer for the PS2 so the graphic designers can use Flash to design and build the menus.

The session is a 50/50 split between myself talking about the production of this system, and Andrew Hamilton, a graphic designer, discussing the trials and tribulations of using it.

How does using Flash for PS2 affect both development workflow and the finished appearance of an interface? Graphic designers like using Flash. They know how to get the best out of it. They can concentrate on making the product look great instead of just acceptable. A good, solid tool at the start of the tool chain makes everyone's life easier.

Flash plays to the PS2s strengths: the PS2 is very good at is throwing loads of polys around the screen. And its vector graphics give the game a really nice TV feel, without using loads of VRAM.

Do you see the "traditional" graphic designer becoming more common in game development? As our industry matures the production values increase. With graphic designers and new-media designers working to make DVD menus, web sites, and CD-ROMs, videogames need to compete with the best of these in production values not only in-game but "out-game" as well. The next generation of videogames needs to take this lead as they have done in real-time 3D graphics.

Besides This Is Football, what recent games' interfaces have impressed you, and why? The Twisted Metal: Black interface has a very clever use of FMV for the background stream and sets the feel of the game fantastically. Though the interface that has impressed the most recently has to be Gran Turismo 3. It has superb communication with the player, a must for a menu system that has everything from selecting which car you want to analyzing the performance of your drive, and industry-leading production values that can be seen right through the game.

How many years have you spoken at GDC? Why do you choose to do so? I attended last year's GDC, and I spoke for the first time at GDC Europe last summer. I enjoyed both experiences very much. GDC really breaks down the barriers between companies and development platforms, and it is a good reminder that we are all part of a big, exciting industry — something I sometimes forget when I'm hiding in the office.

Star Wars Rogue Leader: Rogue Squadron II

When I think back to the development of Star Wars: Rogue Leader, the first thing that comes to mind is time — or rather the lack of it. Never in the more than 13-year history of Factor 5 have we had a project under greater time pressure than this one.

Many might think that Factor 5's history reaches back only as far as 1996, when the company moved to its current location in San Rafael, Calif., just next to LucasArts and Skywalker Ranch.

In fact, Factor 5 was originally formed out of an Amiga hacker group back in Cologne, Germany. In the late 1980s, the Amiga became very popular in Europe, but it didn't have any good action games. It was a port platform, but the machine deserved better; our games, including R-Type and Turrican, were among the first ones to really push the technology unique to the Amiga.

With the Super Nintendo and Sega Genesis/Mega Drive reinventing the console market worldwide, we moved on to these platforms and got into contact with LucasArts, Konami, and Nintendo. During this time, Factor 5 made Super Turrican 1 and 2 and Mega Turrican on the SNES and Genesis, Indiana Jones — Greatest Adventures on the SNES, International Supertsar Soccer Deluxe on the Genesis, and both Contra 2 and Animaniacs on Game Boy.

When the Playstation arrived, we started work on BallBlazer Champions and Star Wars: Rebel Assault 2 for LucasArts. However, the 9-hour time difference between California and Germany soon became a problem with CD-based projects. The Internet wasn't fast enough in the mid-1990s to transfer so much data in any practical fashion. We always had to burn versions to a CD and send them via courier.

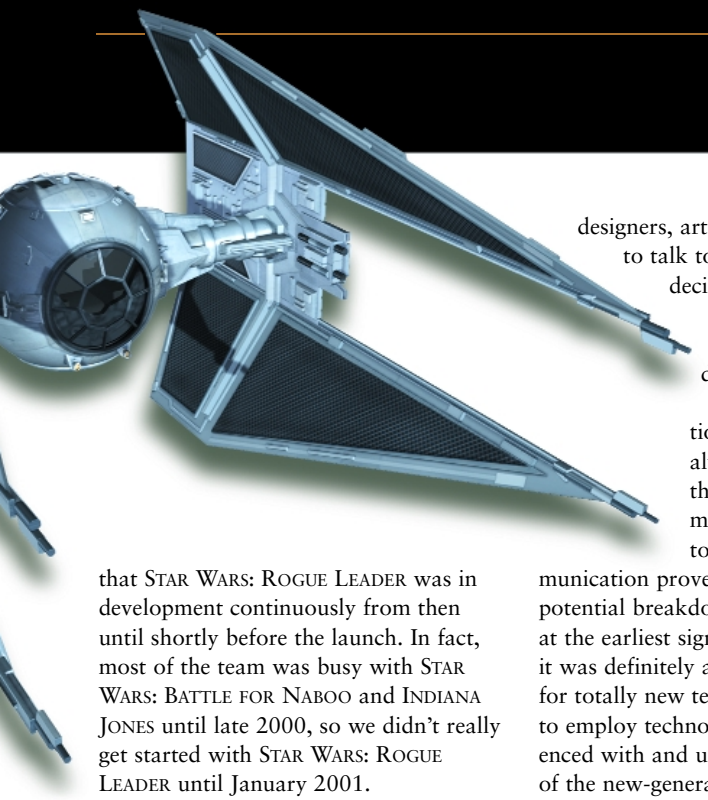
This situation could only go on for so long until LucasArts asked us if we might consider moving the company to the U.S. They offered their help in legal matters, and in May 1996 the American chapter of Factor 5's history began.

After finishing BallBlazer, we moved on to our best-known title before Star Wars: Rogue Leader, the original Star Wars: Rogue Squadron, released in 1998 for the Nintendo 64 and PC. With *Episode I* heading to movie theaters soon after, Star Wars: Battle for Naboo was next, followed by our final N64 game, Indiana Jones and the Infernal Machine.

Those who saw our Star Wars teaser trailer at Space World 2000 might think

In-game screenshot of a scene during the Battle of Endor. The Death Star featured in the background is actually a big sprite and not a 3D model.

THOMAS ENGEL | *Thomas is one of the founders of Factor 5 LLC and has been working in the industry for more than 10 years. As Factor 5's director of technology, his focus has been on the technology behind the gameplay. He has been working on graphics and sound engines, as well as complete games, since the Amiga days. He worked as technical lead engineer on Factor 5's latest game, Star Wars: Rogue Leader.*



that STAR WARS: ROGUE LEADER was in development continuously from then until shortly before the launch. In fact, most of the team was busy with STAR WARS: BATTLE FOR NABOO and INDIANA JONES until late 2000, so we didn't really get started with STAR WARS: ROGUE LEADER until January 2001.

Hitting the Gamecube launch meant being done mid-September 2001 — roughly nine months for a 15-month project.

Fortunately, due to our work on the Space World demo and our involvement in the development of Gamecube's audio system, we already knew a lot of things about Nintendo's new platform. While this gained us the invaluable advantage of having a ready-to-use audio driver and some experience on the Flipper graphics chip, we still had many, many things to test and try out — and pretty much everything we did on the hardware was a first.

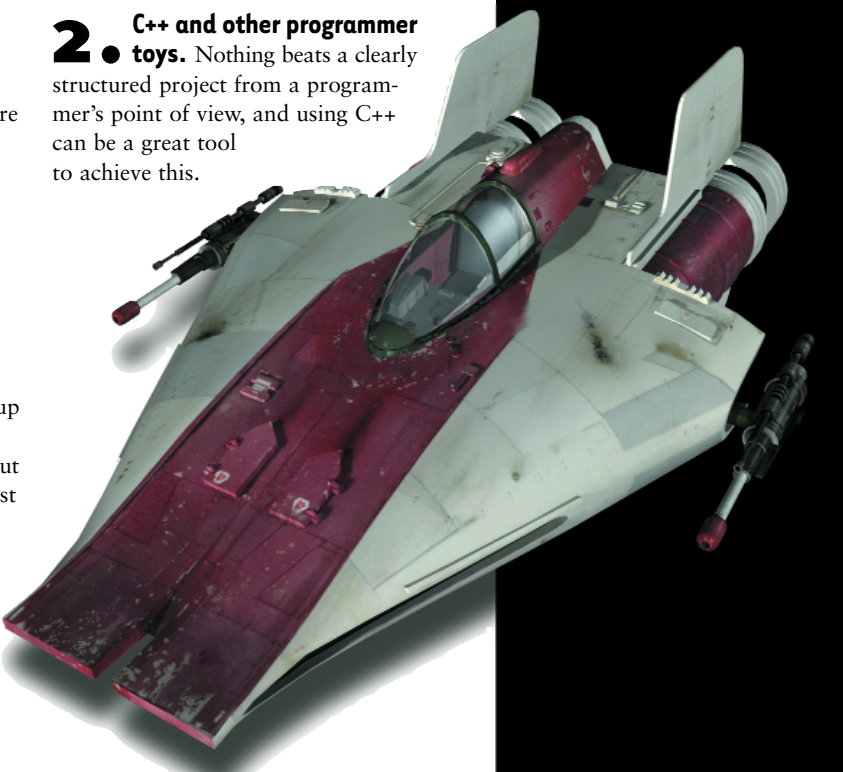
It wasn't long into the project before six- or seven-day weeks became the absolute norm for everybody on the team. And these were not cozy eight-hour days, either.

What Went Right

1. Think first. The need to come up with a workable schedule seems so obvious, and still it does not work out in so many cases. For us, by far the most important step was to come up with a schedule that — even given all the time pressures we were under — was realistic. This included the overall game concept as well as the details of the technical realization. It was absolutely essential to get level designers, artists, and programmers to talk to each other before final decisions were made, and to keep them talking to each other for the duration of the project.

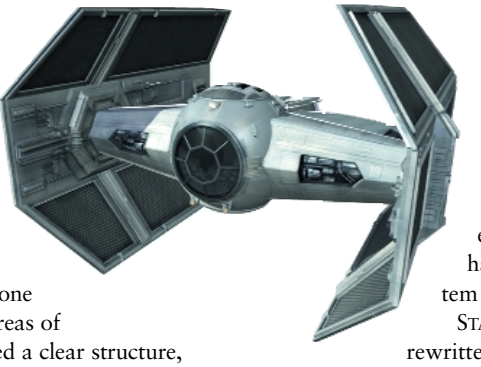
At times communication broke down, but we always managed to rescue the situation quickly. Our most important strategy to maintain good communication proved to be investigating potential breakdowns in communication at the earliest sign. On a technical level, it was definitely a wise decision not to go for totally new technologies, but rather to employ technologies we were experienced with and use the enhanced power of the new-generation hardware to bring everything to a new level. For example, we used a simple height map to represent planetary surfaces, a technique we already had used in STAR WARS: ROGUE SQUADRON and STAR WARS: BATTLE FOR NABOO. Our familiarity with the technology allowed us to concentrate on perfecting it while avoiding potentially catastrophic delays in engine development.

2. C++ and other programmer toys. Nothing beats a clearly structured project from a programmer's point of view, and using C++ can be a great tool to achieve this.



GAME DATA

PUBLISHER: LucasArts
NUMBER OF FULL-TIME DEVELOPERS: 30
NUMBER OF CONTRACTORS: 2
ESTIMATED BUDGET: $3.5 Million
LENGTH OF DEVELOPMENT: 9 months
RELEASE DATE: November 8, 2001
PLATFORM: Nintendo Gamecube
DEVELOPMENT HARDWARE USED: GDEV & 1GHz PC, running Windows 2000
DEVELOPMENT SOFTWARE USED: SN Systems for Gamecube, Slickedit, Maya
NOTABLE TECHNOLOGIES: MusyX 2.0
PROJECT SIZE: 14.2MB of source in 859 files, in-game source data 6.4GB in 10,075 files



We took the time to define up-front the class hierarchies and other guidelines for all the programmers working on the project.

Setting the basic concept for the game in stone very early in the project and assigning clear areas of responsibilities to each programmer introduced a clear structure, and C++, with its protected class members and type checking, helped greatly to keep the structure intact. Since the language itself provided the tools to reinforce the structures defined in the beginning, we were able to minimize the amount of work necessary to maintain orderly source code. This freed up time for the leads to attend to other tasks, and also helped a lot in bringing down the number of reported bugs during testing.

Although we added it in the final month of the project, writing our own virtual memory kernel on top of the OS was another decision that proved to be very helpful in the end. One might ask what virtual memory might be used for on a system that features no writeable mass storage device such as a hard drive.

Dealing with Gamecube's two-part memory architecture, which has 24MB of "fast" (very fast, actually) RAM and 16MB of "slow" RAM that is pretty close to a small ROM cartridge in terms of access and speed, can be a bit of a hassle. This is especially true if one has to make multiple subsystems — implemented by multiple programmers — using the ARAM at the same time. Using the main processor's virtual memory unit, we mapped a section of the ARAM area into the normal address space. We ended up using this dynamic mapping system to avoid having to deal with code overlays by moving code into this virtual memory area, as well as to make access to data in ARAM much easier and more flexible then with manual ARAM DMA transfers.

The time implementing this system was well spent. The last weeks of development saw a number of situations in which we would have lost hours and hours implementing specialized code, but instead the virtual memory system took care of all of them nicely.

3. Scripts without scripts and other level designer magic. The level designers where greatly aided by our proprietary scripting language, CPunsh. CPunsh handles the tasks of a classical scripting language without really being a language in that sense. Rather then implementing a classical computer language, we designed CPunsh as a drag-and-drop-based system of virtual cards. Each card contains a collection of instructions or decision points. The idea behind all this was that we hire our level designers for their expertise in designing fun levels, and not so much for their understanding of programming.

CPunsh's design, while not perfect, in fact helped avoid a lot of bugs in the scripts authored by the level designers and also made them easier to debug if problems occurred. Another bonus was that our level designers already knew the system. It had been part of L3D, our in-house level editor, since STAR WARS: BATTLE FOR NABOO. While we had to add some additional features to support STAR WARS: ROGUE LEADER's new AI system and grander scope, the knowledge that the level designers already had accumulated while using the system earlier proved to be of great help.

STAR WARS: ROGUE LEADER's completely rewritten AI system offered a whole new set of possibilities to the level designers. On the N64 we always had to be overly performance- and memory-conscious. Both STAR WARS: ROGUE SQUADRON and STAR WARS: BATTLE FOR NABOO used close to nothing else but enemies that were running along on predefined splines. This made it quite difficult for level designers to control large quantities of enemies and also make it seem as if the enemies actually would react to the player's actions. With STAR WARS: ROGUE LEADER, this system got its long-overdue revamp. In this title, enemies are still guided by splines, but most of the action is controlled by flocking and other algorithms and is highly aware of the player's actions. The added creative freedom for level designers was truly a great asset.

4. Art: Painting by polygons. One thing our work on the original Space World demo really helped with was to get a thorough understanding about the basic art requirements of this title. The demo offered us a test run in terms of getting artwork out of Maya into the run-time engine. Only the basic animation and geometry pipeline developed for the Space World demo ended up in the final product, but this proved to be a key asset in speeding up the development of the shader data path later on, since we didn't have to start entirely from scratch.

Both the programmers and the artists had a clear understanding of what they wanted, and the specifications for geometry in particular were clear long before the bulk of the team moved over to the development of STAR WARS: ROGUE LEADER. This technical groundwork, together with the exceptional work of all the artists on the team, helped STAR WARS: ROGUE LEADER achieve the visual quality one sees in the final product.

One of the strengths of Gamecube's hardware is multi-texturing. Using well-understood techniques for our geometry representation and generation, we decided to concentrate our efforts on the texturing aspect.

With respect to craft models, this was definitely the right decision. The classic *Star Wars* designs don't lend themselves too well to the modern ways of compressing and refining geometry representation, such as subdivision surfaces or NURBS, due to their boxy and angular structures. To get accurate representations of these models, we had to rely less on technology and more on first-class modelers.

For the landscape, which was represented by a height map, the texturing was the single most important aspect of all. Only with multi-texturing was it possible to achieve the organic and natural look we were going for. The landscape texturing consists of multiple layers of repeating, general patterns. The trick was to combine all these layers with what we called "mix-maps," a set of simple grayscale textures that defined how the

LEFT. The Ison Corridor fight, featuring volumetric fog. TOP RIGHT. A Snow Speeder is busy "cabling" an ATAT. A physics simulation drove the cable's movement. BOTTOM RIGHT. An X-Wing aproaches the grounded Star Destroyer on Kothlis. Great care was taken to get the water shaders just right.

different types of patterns were to be combined. To add even more flexibility, we also allowed the mixmaps and patterns to be rotated against each other. Besides offering good looks, the use of mixmaps also gave the textures a small memory footprint, since we could easily hide the repetition of the patterns with clever setups for the mix-maps. Bump and detail maps finished off the effect.

All these texturing technologies were integrated either into L3D, our in-house level editor, or Maya's shader controls. This way, the artists and level designers had an easy-to-use interface in which to create all the texture artwork.

Some issues remain to be solved for our next game. For example, there was no fast and easy way for the artists to preview their work on the real hardware. Unfortunately things look quite different on a PC monitor than on a 640×480 TV display, but we were still pleased with the results.

5. Making some noise. With regards to audio, we had a good start. Factor 5 had an important role in the development of Gamecube's audio hardware, and we developed the MusyX audio system in-house. Because of this, we were able to build the audio part of STAR WARS: ROGUE LEADER on a solid, fully understood API and tools.

On the creative side of things, it helped a great deal to have access to the LucasArts and Lucasfilm archives. While a lot of effects and post effects for voice recordings had to be redone and redesigned by our sound effects designer, having access to this data was very important in keeping things sounding authentic. We are in the lucky position of having two dedicated and very experienced sound designers at Factor 5, assets that can't be overvalued if one has to work on a tight schedule.

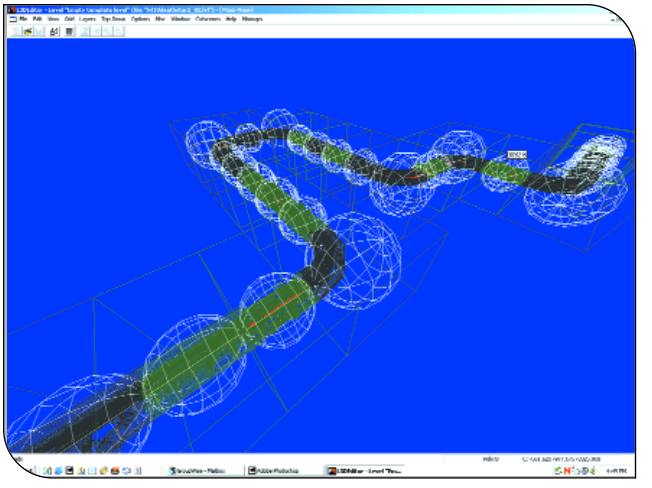
We also implemented a little tool that allowed the sound effects designer to mix the audio much like a sound designer in a movie would do. Using this tool, the sound designer is able to manipulate most parameters of sound effects in the game while the game is running. Since mixing sound effects is as important as designing their basic characteristics, this tool was truly worth the work spent on it.

STAR WARS: ROGUE LEADER was the first game ever to feature a Dolby Pro Logic II (DPL2) surround sound encoder. When Dolby showed us the results one could produce using DPL2 while staying 100 percent compatible with the widespread Dolby Pro Logic system, we were truly amazed. Because we started with such a solid audio base, we could invest the time to develop our own DPL2 encoder for use on Gamecube.

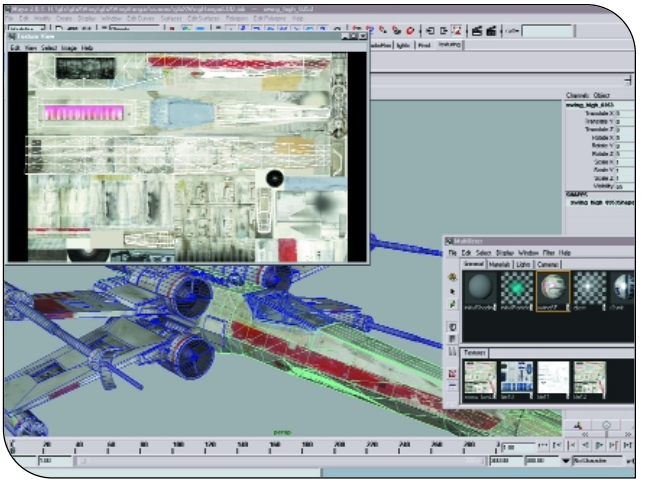
What Went Wrong

1. Make data and coffee? Some data conversion runs took forever. Since everybody in the team to some extent depended on the data conversion, the impact of long data conversion runs multiplied quickly.

There were multiple reasons for the slow speed, most of which can be traced back to two fundamental problems. First, we did not implement a global-caching system for converted data. Because of this, multiple people would convert the exact same data on their local hard



The tunnel sequence of the fight for the second Death Star as seen in L3D. The boxes are added to ease the culling process, while the spheres control certain aspects of the gameplay like triggering of events or lights.



An X-Wing modeled in Maya. We used tightly packed texture sheets wherever we could to minimize the overhead introduced at run time due to different material setups. It proved much faster to go for a more complex global material setup than for multiple simpler ones.

drives while the converted data was actually already distributed around in the company. For some data types, such as textures and shadow maps, we later introduced caches on the server, but a more global scheme would have helped greatly.

The second major problem had to do with interdependencies between different data types and even program code. Interdependencies between data resulted in rebuilding of a lot of data, even if just small amounts of data changed. This could have been avoided with a different setup for the data, and perhaps by the introduction of some kind of linker for the data conversion. All this never happened for this project. Nobody had the time even to plan for something at the point when the problem became obvious. From this we learned one thing: Do not underestimate the amount of data one has to handle in a game for the (now) current generation of game consoles.

2. Cutscenes. As we did in our earlier titles, we wanted to avoid using FMV for cutscenes, to avoid breaking the continuity and style of the game. The playback of these animations was handled quite elegantly by our standard run-time animation system, which we used for all types of animation throughout the game. The internal structure of this system closely resembles Maya's animation system, which made it very flexible and straightforward to control from within that tool. It also was really easy to use at run time. This is where the fun ended, however, and things started to get ugly.

On the programming side, we had to spend a lot of time organizing the data flow. We wanted to make the amount of data to be loaded or kept in memory for cutscenes minimal, which meant recycling data that was present in the level data currently loaded. This sounds easier — and we fell for it, too — as it is actually was. However, our main problem was that we weren't able to begin implementation of the cutscene playback until pretty late into the project, and the programmers responsible for the task were confronted with a data structure that was pretty much set in stone. This is the one area in which our planning at the beginning of the project did not quite work out.

In addition, the animators could not start until relatively late in the project. This was due to both the unfinished system implementation as well as their simply being occupied with other tasks. This time frame put the animators under a lot of pressure. To make things even worse, they had to suffer quite a bit under the slow data conversion. The cutscene data included so many cross references into other types of data that changing a cutscene frequently meant rebuilding major parts of the data set.

Another thing that slowed us down was the fact that we could only export part of a level's data into Maya as a reference for the animators to work with. In particular, the long cutscenes in the Hoth level, featuring many close-to-ground camera moves, presented a challenge, since the height-map geometry export into Maya always used the lowest level of detail and hence didn't represent enough detail for such tight corner moves.

Previewing music and sound effects for cutscenes proved to be another hassle. The effects and music were triggered from within Maya but could not be previewed there, which meant going through the data conversion step each time we wanted to test things.

More preview capability and a faster data conversion would have solved our problems. It also wouldn't have hurt to have more time, a luxury we simply couldn't afford this time around.

3. What's that engine glow doing on the nose cone? Little things can cause big problems when time is as much a factor as it was with this project. The method we used to add visual effects, like engine glows to the in-game craft models, introduced a high dependency between the code versions used to edit new levels and preview models, and the model data

itself. A simple change to the model within Maya could trigger engine glow to appear in the completely wrong spot.

This made it close to impossible for the artists to judge whether they actually did something wrong while exporting the data or whether this was just one of the cases where everything was actually in order and only the code version needed an update. This situation was truly counterproductive in the last weeks of production.

We will have to remove this dependency completely from our next game's data path. To some degree, this will be a side effect of our effort to speed up the general data conversion by removing data interdependencies.

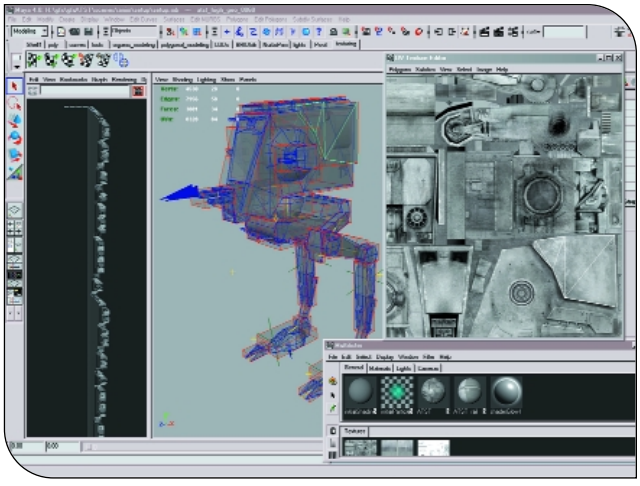
4. "I've got some new songs." While we had pretty good turnaround times getting completely new sound effects into the game — as much as in-game material was concerned — the data path and the way the music was linked into the game did not allow for any fast preview cycles for streamed or MIDI-based pieces of music.

Due to time constraints, we had been unable to implement tools that would have allowed adding or changing pieces of music without rebuilding the program code. Furthermore, most pieces of music are triggered from scripts designed by level designers, which meant we always had to coordinate three people in the process: a programmer, a level designer, and the musician.

In the end we coped with the problem to some extent by simply setting up a schedule defining when a new music update would be done. The musician was able to audit his score without the integration into the game on a prototyping tool at any point in time and with minimal turnaround times. The schedule made the problem somewhat manageable for all parties involved, but it was still a far cry from being a good working environment.

5. L3D: Trusted, proven and, well, a bit rusty. L3D, our in-house level editor, had been around for many years when we started STAR WARS: ROGUE LEADER. The good thing about this was that the level designers knew the tool and it had gone through a lot of iterations of improvements during the development of STAR WARS: ROGUE SQUADRON and STAR WARS: BATTLE FOR NABOO. But we also knew that it wasn't exactly designed to deal with data sizes as they regularly pop up with this new generation of consoles. We simply didn't have the time to program a new tool, since level designers had to start their work early. We just hoped for the best.

Some things got very slow as things got very large in terms of memory. For the most part we could get things to a workable level again by running the tool on relatively high-end systems. Other things just had to be endured by the level designers. The situation was far from perfect. Despite its shortfalls, however, we couldn't have made it without this old, trusted, and slightly outdated tool. Still, a new, streamlined tool would have been much faster and more user-friendly, which in turn would have given the level designers more time to actually design things. As a logical consequence, this is what we are currently working on. With a bit more time on our hands for finalizing our next title, we took the opportunity to take the things we learned from L3D and rewrite a new tool from scratch.



An ATST's texture sheet and structural setup displayed in Maya. To get as close as possible to the look of the animation in the movies, we chose to use relatively complex animation setups for all major animated objects. The Gamecube's hardware coped well even with large numbers of such objects, as can be seen in the Battle of Hoth level.

Final Thoughts

It was a ride. It was also a stressful, demanding period in all of the team members' lives, but still an exciting experience that probably nobody on the team would have missed for anything in the world — so long as it's guaranteed not to be repeated anytime soon.

Never have I seen a team so dedicated to a game project than the STAR WARS: ROGUE LEADER team. Everybody had just one aim: make the best game possible. Working in a small company without any major bureaucratic overhead also definitely helped to keep everybody going at full speed.

We couldn't have done it without help, though.

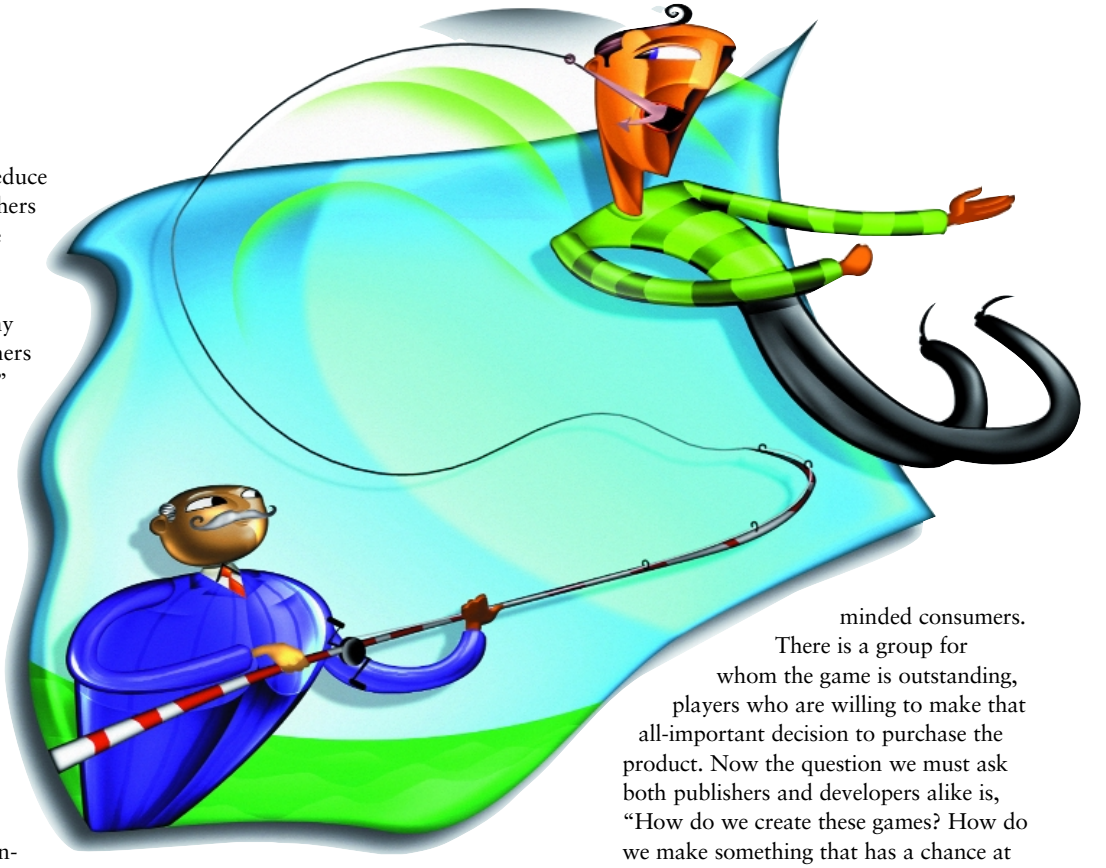
The full and unconditional support from LucasArts was essential in keeping everybody focused on the task. You don't want to worry too much about your relationship with your publisher when you have a plate as full with other tasks as we had.

Also, working on a tight schedule made having a good test department all the more valuable. LucasArts' test department really supported us, even moving into our offices to further optimize the process. Nothing beats the productivity resulting from testers always being available on a moment's notice.

Last but not least, we enjoyed the professional and dedicated support of both Nintendo Technology Development and Nintendo's Software Developer Support Group. I can't stress enough how important their input was in getting us up to speed on the new hardware.

Looking at the response the game has triggered so far, we seem to have done a pretty good job. In the end, only time will tell whether the game lives up to what people expected from it.

Let Developers off the Hook



In their desire to reduce risks, some publishers are driving a knife into the heart of our industry.

I can't tell you how many times I've met with publishers who talk about "the thing" or "the hook." You see, a marketing or sales department gets the idea that some single element has made a particularly popular game sell, and that they've identified that single element. They know what the thing is. DUKE NUKEM is a great example of this. Many of the marketing and sales guys that I spoke with had played the game, and each believed that they'd identified the one single element that made it a success. Some decided that it was that the fact that the levels could be destroyed. Others chose the wise-cracking humor that Duke spat out during the game. The point is, different people chose completely different things, each believing that they'd identified "the thing" that was the single reason for the game's success.

And they were all wrong. It wasn't just one thing that made the game great.

Crediting a game's success to a single game element is analogous to the story of the five blind men describing an elephant: The first only touches the elephant's leg and says, "It's a tree." The second touches the side and says, "It's a wall." The third touches only the trunk and says, "It's a snake." The fourth feels the tusk and thinks it's a spear, while the fifth, touching an ear, says it's a giant fan. Each believes that he has described the entire elephant, that he has discovered the thing that an elephant is. But an elephant, like a game, cannot be fully defined by only one of its attributes.

Publishers need to get a better perspective on the one feature that is common to all successful games. The thing that virtually all successful titles share is that they are good games. No one game is all things to all players, but every successful game appeals to some large body of like-minded consumers. There is a group for whom the game is outstanding, players who are willing to make that all-important decision to purchase the product. Now the question we must ask both publishers and developers alike is, "How do we create these games? How do we make something that has a chance at being accepted by some portion of the gaming community?"

The answer is simple: Make sure there is always at least one person on the development team who absolutely loves the game being made. Find companies that have created games that had the attention to detail that can only come from a true caring for the product. The reason that DUKE NUKEM was great was that the team cared about it. The reason there were destructible parts in the levels and you could play pool on the tables in the bars was because they wanted to make the game perfect, not because the team was trying to fill in every possible

continued on page 71

Illustration by Peter Lacalamita

continued from page 72

bullet point or check off items on a list. Games, like all artistic creations, be they music or movies or paintings, must be built by people who care about them. Publishers need to look for teams that have done great work large or small. The goal should be to find people who care.

In order to create great games, developers don't need to focus on technology, but on controls and the user interface. Make sure that you like to play your game, and then keep working on it until you love it.

And I would tell a publisher to keep looking for those teams that can make great games, not just hit bullet points and target system requirements. Great programmers and great artists can build on any system that you ask them to build on. Just make sure that they know how to care about what they're doing.

There are stories of wonderful publishers who've given their developers the time, resources, and trust needed to create great products. Regrettably these stories are few and far between. Until there is more agreement between developers and publishers about what constitutes truly great games and how to make them, we will hear more and more stories about how a great game was rejected by five or 10 publishers before someone could finally realize its potential.

LARS BRUBAKER | *Lars is currently the CEO of Reflexive Entertainment Inc., a company he co-founded four years ago. During this time Reflexive has created such titles as* SWARM, STAR TREK AWAY TEAM, ZAX: THE ALIEN HUNTER, *and* RICOCHET. *Previously, Lars was producer and senior software engineer at Logicware, where he designed, produced, and programmed* ASTROROCK *and produced and co-programmed* DEFIANCE. *Before that, Lars worked at Interplay in roles ranging from quality assurance to lead programmer and producer.*