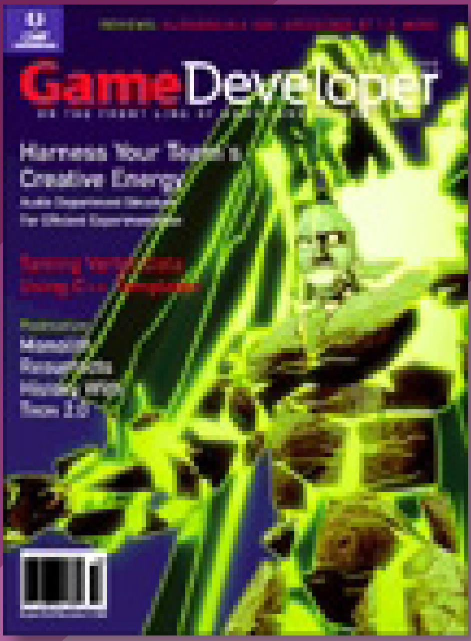
gd

GAME DEVELOPER MAGAZINE

OCTOBER 2003



UBM Tech

Return

GAME PLAN

LETTER FROM THE EDITOR

Creativity: A Friend in Need

Regular readers of this column may have noticed an alarming trend: that I am not an alarmist. Upon ascertaining those frequent reports of the game industry's imminent demise, I, as Mark Twain did upon hearing reports of his own death, find them greatly exaggerated.

The preponderance — by which of course I mean broad consumer success — of license-based games and sequels is an area of grave concern to many developers who would prefer to develop their own IP and for those who rightly see content diversity as key to the game industry's long-term future. But this trend toward flashy licenses and astronomical production values need not sound the death knell for creativity in game development, unless you mean to equate the idea of "the creative process" exclusively with "total chaos." And, based on our industry's history, chaos may be something from which game development would do well to distance itself.

Fortunately, there are more productive and replicable ways to leverage creativity that game developers are beginning to explore with encouraging results, which you can read about this month. Yes, publishers and their risk-averse henchmen are our creative nemeses, but developers themselves must assume some of the guilt for overreliance on designs and implementations strictly by virtue of their having worked in the past. Ideas born of creative incest lead to sickly, hemophilic dead-ends and stagnant evolutionary backwaters.

With larger teams across art, programming, design, production, and audio, the need for structure prevails over even the most intangible creative pursuits. When so many people get involved in a collaborative endeavor, the tenacious specter of chaos is never far removed. Structuring creative processes to leverage the output potential of randomness and free-form thinking without the operational chaos it induces is the subject of Rob Bridgett and Wolfgang Hamann's feature on managing audio departments, which begins on page 28. Beyond the fact that in-house and out-of-house audio departments' roles on development projects is evolving to a greater level of integration with the rest of the development team than ever before, the lessons Bridgett and Hamann offer can just as well apply to other creative realms in game development and beyond.

Also this month, Masaya Matsuura, musician and creator of PARAPPA THE RAPPER, UM JAMMER LAMMY, and MOJIBRIBBON, echoes the benefit of jam session–style brainstorming in game development in his Sound Principles column on page 24. The world's greatest musicians have long sought new inspiration from their bandmates, colleagues, even adversaries through experimental performance within commonly understood foundational structures of music. Game developers feeling constricted creatively by technology or existing IP should consider what musicians can accomplish in a jam session with 12 notes and a handful of recognizable time signatures.

Historically game developers have not looked to the industry's audio professionals for technological guidance or substantial design input, but there is much to learn from the process behind great sound and music that has clear implications specifically for visual and design development in addition to audio. Much of musical experimentation is based on structured iteration of theme and variation. If you're stuck creatively with a theme not of your choosing, it's time to refocus on the experimental processes that lead to the variations.

Happy Trails, Hayden. As revealed last month, this issue marks Hayden Duvall's last Artist's View column for *Game Developer*. We wish Hayden the best and look forward to introducing his successor next month.

Jennifer

Jennifer Olsen
Editor-In-Chief

GameDeveloper
www.gdmag.com
600 Harrison Street, San Francisco, CA 94107 t: 415.947.6000 f: 415.947.6090

EDITORIAL
Editor-In-Chief
Jennifer Olsen *jolsen@cmp.com*
Managing Editor
Everard Strong *estrong@cmp.com*
Departments Editor
Jamil Moledina *jmoledina@cmp.com*
Product Review Editor
Peter Sheerin *psheerin@cmp.com*
Art Director
Audrey Welch *awelch@cmp.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Jonathan Blow *jon@number-none.com*
Hayden Duvall *haydend@3drealms.com*
Noah Falstein *noah@theinspiracy.com*
Advisory Board
Hal Barwood LucasArts
Ellen Guon Beeman Monolith
Andy Gavin Naughty Dog
Joby Otero Luxoflux
Dave Pottinger Ensemble Studios
George Sanger Big Fat Inc.
Harvey Smith Ion Storm
Paul Steed Microsoft

ADVERTISING SALES
Director of Sales/Associate Publisher
Michele Sweeney e: *msweeney@cmp.com* t: 415.947.6217
Senior Account Manager, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 828.350.9392
Account Manager, Northern California & Midwest
Susan Kirby e: *skirby@cmp.com* t: 415.947.6226
Account Manager, Western Region & Asia
Craig Perreault e: *cperreault@cmp.com* t: 415.947.6223
Account Manager, Recruitment
Raelene Maiben e: *rmaiben@cmp.com* t: 415.947.6225
Account Representative, Target Pavilion & Education
Aaron Murawski e: *amurawski@cmp.com* t: 415.947.6227

ADVERTISING PRODUCTION
Advertising Production Coordinator Kevin Chanel
Reprints Terry Wilmot t: 516.562.7081

GAMA NETWORK MARKETING
Senior MarCom Manager Jennifer McLean
Marketing Coordinator Scott Lyon

CIRCULATION
BPA INTERNATIONAL
Game Developer is BPA approved
Director Kevin Regan
Manager Peter Birmingham
Assistant Manager Lisa Oddo

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

EDITORIAL FEEDBACK
editors@gdmag.com

CMP MEDIA MANAGEMENT
President & CEO Gary Marshall
Executive Vice President & CFO John Day
Executive Vice President & COO Steve Weitzner
Executive Vice President, Corporate Sales & Marketing Jeff Patterson
Chief Information Officer Mike Mikos
President, Technology Solutions Robert Faletra
President, Healthcare Media Vicki Masseria
Senior Vice President, Operations Bill Amstutz
Senior Vice President, Human Resources Leah Landro
Vice President & General Counsel Sandra Grayson
Vice President, Group Publisher Applied Technologies Philip Chapnic
Vice President, InformationWeek Media Network Michael Friedenber
Vice President, Group Publisher Electronics Paul Miller
Vice President, Group Publisher Software Development Peter Westerm
Corporate Director, Audience Development Shannon Aronson
Corporate Director, Audience Development Michael Zane
Corporate Director, Publishing Services Marie Myers

CMP United Business Media
GamaNetwork

Return

SAYS YOU

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

Norrath, 90210

We wanted to respond to Damon Watson's article "MMORPGs: Perfect Game or Dangerous Addiction?" (Soapbox, July 2003) and say that we strongly believe MMORPGs are dangerous addictions. Our 22-year-old son plays EVERQUEST from the time he rises (late afternoon) until he goes to bed (middle of the night). He was enrolled in college classes for several semesters, but he failed to finish because he was not attending. All he cares about is EVERQUEST.

We feel he is escaping his real-world problems and his responsibilities of finishing school and finding employment. From personal experience and in our opinion, MMORPGs are unhealthy and definitely addictive.

Steve and Rose Ussery
Hollister, CA

PRODUCT REVIEWS: LCD DISPLAYS, THE SFX KIT, VISUAL SLICKEDIT 8
CMP
JULY 2003
Game Developer
ON THE FRONT LINE OF GAME INNOVATION
Staring Down the Enemy: Computer-Player AI
Content Optimization Techniques
Postmortem: Big Huge Games' RISE OF NATIONS
Online Game Addiction: Realities and Liabilities

From personal experience and in our opinion, MMORPGs are unhealthy and definitely addictive.

Hackneyed. Overused.

I just read Jennifer Olsen's list of words that make game titles forgettable ("Sequels 2K3: Beyond the Return of the Sequels," Game Plan, July 2003), and I would like to add the following:

Phoenix. War. Xeno. Lost. Quest. Of (as in Call OF Duty, Medal OF Honor, Men OF Valor). Oh yeah, might as well add Valor, Duty, and Honor while we're at it.

Ralph A. Barbagallo III
Flarb (via e-mail)

Everybody Loves Hayden

I really enjoy reading the Artist's View articles written by Hayden Duvall. He emphasizes the importance of using proper aesthetics in game art for the sake of quality and enjoyment. He also focuses on the psychological effects a game can have on players, and how a game artist can, and should, use them to his or her advantage in order to make the game more enjoyable.

If a game is going to have good replayability, then psychology should be emphasized, more so than math. Challenging our minds and having an emotional experience in the process is what ultimately makes the game fun to play and keeps players coming back.

I praise Mr. Duvall for his sincere and intelligent approach to game art.

Roberto Moreno
via e-mail

Departments Editor Jamil Moledina replies: *Sadly, the issue you're holding contains Hayden's last entry in the Artist's View column. However, our November issue marks the debut of a writer we believe is poised to develop a following of his own.*

Simplify Your Tables

In Jay Lee's "Data-Driven Subsystems for MMP Designers" (August 2003), he never discusses the relational/object divide. I have seen many projects fail because they've mapped too closely from objects to relational tables, and have concluded that defining a solid data-access abstraction layer and allowing your DBA/DBE to tune your database is the most effective way to gain from your database.

For instance, when discussing relationships, he speaks of the "1-to-1" relationship — this is an ideal candidate for normalization into a single table, even though it may not correlate directly to any "entity." Accessing the new normalized table requires one less join than the previous data model, and performance is enhanced. These tuning steps can add up to a big performance win, especially with well-constructed indices.

Michael Lanzetta
via e-mail

Rule 401

In reference to Noah Falstein's Better by Design column, I have a rule of my own to propose to ensure game quality:

Imagine yourself climbing a mountain. The person holding the rope has just spent the entire past week playing your game. Do you feel safe telling him you designed that game?

edA-qa mort-ora-y
via e-mail

Noah Falstein replies: *Neatly put. Although personally, I don't know that I'd trust my life to a climbing partner who'd spent the week training for the climb by sitting in front of a screen. Even if he was playing a vintage emulation of* CRAZY CLIMBER. *Maybe especially not then.*

E-mail your feedback to editors@gdmag.com, or write us at *Game Developer*, 600 Harrison St., San Francisco, CA 94107

Return

INDUSTRY WATCH

KEEPING AN EYE ON THE GAME BIZ | *jamil moledina*

Sony spills PSP specifics. Sony disclosed that its upcoming handheld device, the PSP (Playstation Portable), will be built around a MIPS R4000 32-bit core, enabling 3D graphics. The device will feature embedded wireless LAN functionality, optical disc media, 7.1 channel audio, and a 16:9 widescreen LCD screen. Sony expects to position the PSP directly against Nintendo's Game Boy Advance, although the device will also play music and movies. Sony plans a global simultaneous release of the PSP in late 2004.

Microsoft immerses Immersion in cash. Microsoft agreed to pay Immersion $35 million to resolve Immersion's patent infringement claim. Under the terms of the settlement, Microsoft gains licensing rights to Immersion's haptic technology, and also an unspecified equity stake in Immersion.

THQ brings Rare games back to Nintendo. THQ signed a deal with Microsoft to publish games developed by Microsoft's Rare game studio on Nintendo's Game Boy Advance. THQ plans a fall release for the first title based on this partnership, BANJO-KAZOOIE: GRUNTY'S REVENGE.

THQ bridges Rare and Nintendo, publishing Rare titles such as BANJO-KAZOOIE: GRUNTY'S REVENGE for Game Boy Advance.

Game Boy juggernaut overtakes Gamecube. Nintendo posted a Q1 profit of ¥11.5 billion (US$95 million), a 5 percent increase over the previous year. The company reported that it sold 3.24 million Game Boy Advance and Game Boy Advance SP units worldwide, but only 80,000 Gamecube units for the quarter.

Nvidia invades mobile chip market. Nvidia began a purchase of MediaQ, a graphics and I/O chip maker for mobile and handheld devices. Both companies' boards have approved the deal, valued at $70 million. Nvidia expects the transaction to be completed in Q3 of its fiscal year.

ATI boxes out Nvidia? ATI signed a deal with Microsoft to provide the graphics system for the next Xbox. Although Nvidia provides graphics chips for the current Xbox, the company entered arbitration with Microsoft last year to resolve the price of those chips.

Midway loses shell game. Midway Games failed a covenant of its $15 million credit line, resulting in the line being terminated. In an SEC filing, Midway admitted that its failure to comply with requirements related to minimum stockholders' equity and net worth was due to a $23.05 million write-down of capitalized product development costs. Midway also said it received a $4 million payment from its former parent company, WMS Industries, related to tax sharing and separation agreements.

Send all industry and product release news to news@gdmag.com.

THE TOOLBOX

DEVELOPMENT SOFTWARE, HARDWARE, AND OTHER STUFF

Splutterfish releases Brazil 1.2, Shave and a Haircut. Splutterfish's latest rendering tool, Brazil Rendering System Version 1.2, includes distributed rendering, advanced shadow plug-ins, and an advanced skin shader. In conjunction with Joe Alter, Splutterfish also released Shave and a Haircut, a CG hair grooming, dynamics, and rendering system. Both tools interoperate with Discreet's 3DS Max animation software. Brazil Rendering System Version 1.2 starts at $750, while Shave and a Haircut has a retail price of $485. www.splutterfish.com

Bodypaint 3D 2 now available. Maxon Computer released Bodypaint 3D 2, a 3D texture painting application that works with 3DS Max, Maya, and Lightwave. New features include projection painting, raybrush, improved data exchange with DCC programs, and better OpenGL rendering. Bodypaint 3D 2 sells for $645. www.maxon.net

Digital Anarchy releases texture plug-ins. Digital Anarchy released a set of three plug-ins for Adobe Photoshop called Texture Anarchy. The set can be used to create fractal-based procedural textures for use as seamless texture maps, borders, and backgrounds. Texture Anarchy includes customizable light sources, gradient tools, and 38 types of noise. It is available for $129. www.digitalanarchy.com

UPCOMING EVENTS CALENDAR

INDIE GAMES CON 2003
WILD DUCK BANQUET HALL
Eugene, Ore.
October 10–12, 2003
Cost: $150–$195
www.indiegamescon.com

MOBILE GAMES 2003
MELIA AVENIDA AMERICA
Madrid, Spain
October 22–24, 2003
Cost: £699–£1,699 (+VAT)
www.ef-international.co.uk/index.cfm?conference=2220

Return

NXN's Alienbrain Studio 6 Integrator SDK

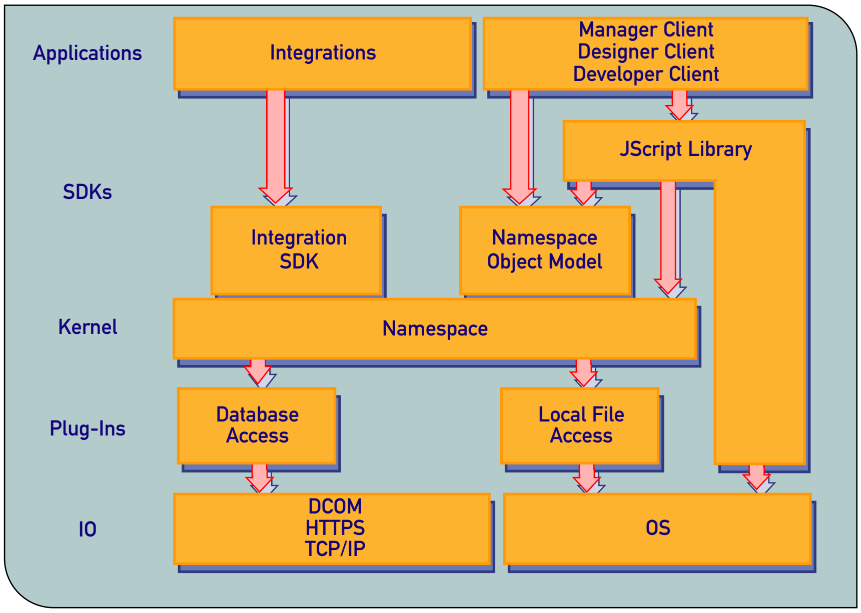
by jeremy gordon

Asset management is a huge challenge facing content creation teams today. Every development team seems to have their own custom content creation tools and techniques. The fact that Alienbrain Studio's functionality can be programmatically accessed through the included software development kit (SDK) allows teams to fit Alienbrain Studio to their custom tools and techniques, rather than the other way around.

Alienbrain Studio is based on a client/server architecture. Graphical user clients (manager, designer, or developer) connect to the server over one of the supported transport types: DCOM (the default), HTTPS, or TCP/IP.

It's possible to extend the various graphical clients with new views using JavaScript and XML. In addition DHTML can be used to create custom preview panels (in the graphical clients supporting them), usually in conjunction with a custom-authored ActiveX control. It's also possible to extend the server-side thumbnail generator to understand new formats through the use of a purpose-developed DLL. None of these customizations typically involves the Integration SDK, but I mention them here for completeness.

Custom applications (or "Integrations"), such as a level editor or proprietary build system, must link with the Integrator SDK to access the Alienbrain Studio "Namespace." The Namespace provides all core client functionality such as database and local file access. Everything from items and properties under version control to local files and folders live in the Namespace.



How a client/server architecture works inside Alienbrain Studio.

The SDK includes HTML format documentation, sample code and C++ header files, and libraries which implement a set of classes for manipulating the Namespace. Both debug and release versions of the libraries are available, which greatly facilitates tracking down bugs.

There are several well-illustrated tutorials that provide a walkthrough of the basics required for writing simple Integrations, but more advanced examples are missing. The classes are logical and fairly well thought out, but not as abstract as they could be, and in some cases they are a little clunky. Classes such as CNXN Smart Integrator aggregate the functionality of other, more specialized classes, allowing authors to pick and choose to what degree of granularity they want to use the SDK.

Along with API calls for the basics, including get latest, check-in, checkout, and the iteration of version history, the SDK supports the ability to receive "push" style notifications when interesting events happen in the Namespace. An Integration can learn about most events both right before and right after they occur. Handling events in an Integration is pretty easy; user classes are derived

JEREMY GORDON | *Jeremy Gordon is the president and CEO of Secret Level, a boutique game developer located in San Francisco.*

★★★★★	excellent
★★★★	very good
★★★	average
★★	disappointing
★	don't bother

from the CNXN Event Target class and use Microsoft Foundation Classes (MFC)-style message map macros to wire Alienbrain Studio event names to custom member functions.

A word on NXN's developer support: it's top-notch. Our e-mail queries have received automated responses immediately with an average resolution time of typically only a few hours. NXN also maintains a password-protected support web site with downloads and additional information.

One major downside to the SDK is that it is currently only available on the Windows platform, so Linux users will have to make do with the command line tool for now. In addition to a Linux version of the SDK, a robust server-side SDK definitely makes my wish list.

ALIENBRAIN STUDIO 6 INTEGRATOR SDK ★★★★

STATS

NXN
Venice, Calif.
(310) 393-8535
www.alienbrain.com

PRICE

$3,999 and up

RECOMMENDED CONFIGURATION

One server per team, one client per team member plus sufficient client access licenses to accommodate all users. Server and client choices depend on the size of your team.

PROS

1. Provides deep access to NXN Alienbrain Studio functionality.
2. Top-notch developer support.
3. API is stable across Alienbrain Studio versions.

CONS

1. Only available on the Windows platform.
2. PCs running applications built with the SDK require a client license.
3. Lacks sample code demonstrating advanced API features.

With version 7 of the software coming soon, users will be relieved to know that custom Integrations will not require source code–level changes to remain compatible.

Another downside of the SDK is that an Alienbrain Studio client must be installed and licensed in order to run an Integration; just having the server isn't enough. The SDK comes free with Alienbrain Studio — but, as discussed in previous reviews in this magazine, depending on the configuration, the client (and server) can set you back a little more than competing version control packages. That said, if you can eschew the fancy features of the manager and designer clients, the developer client pricing is actually quite competitive with other version control packages, and the Alienbrain Engineer server (which allows connections from developer clients only) is free of charge.

Overall, the SDK is a solid performer, allowing deep access to the Alienbrain Studio client functionality. One of the biggest reasons to implement an Alienbrain Studio installation is to provide dependable, artist- and designer-friendly digital asset management. The Alienbrain Integrator SDK enables developers to extend this ideal to their own custom content creation tools.

Tricks of the 3D Game Programming Gurus

by André LaMothe

reviewed by jeremy jessup

In his latest book, *Tricks of the 3D Game Programming Gurus*, André LaMothe tackles the development of a 3D software engine in a systematic and instructional manner. The book is a little over 1,600 pages, comes with a companion CD, and retails for $59.99. The book captures the complexity of graphics programming to a tee. LaMothe doesn't shy away from difficult material and provides excellent reference materials to help supplement the text.

Writing a graphics engine in software may not seem all that sophisticated, but it is an excellent way to approach computer graphics. By writing specific functions that are typically abstracted by a platform-specific API (such as DirectX), LaMothe focuses on the underlying theory and provides the reader a conceptual framework that is easily adapted to various targets as the need arises.

While this book is the second volume in the *Tricks* series, it is not essential to have read the first book. To handle the 2D graphics, audio, and input, LaMothe builds the 3D engine on top of the 2D engine in the first book with DirectX 7.

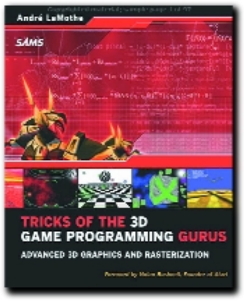
The first section introduces DirectX, the basic game structure, and the previous library's functional interface. To optimally build the 3D engine, LaMothe abstracts the DirectX and Win32 code by encapsulating the computer interface to a set of three libraries which handle window construction, input, and audio. The book adequately describes the basic foundations necessary to use DirectX and Win32 without dwelling on many of the specifics, since the focus is on the 3D engine.

The second section begins with linear algebra and trigonometry. The math section spans over 100 pages and forms the basis of the math library described in the subsequent chapter. Having most of the fundamental groundwork in place, LaMothe begins to develop the pipeline for the 3D engine. From local to world transform to projection, the substeps necessary for rasterization are described in detail. In order to read external model data, several functions are developed to parse the output of the modeling tools which are included on the companion CD. By the end of the section, the engine is able to render in wireframe.

LaMothe starts the third section of the book adding critical enhancements: lighting, texture mapping, clipping, and a depth buffer. Starting with the mathematical background, each topic is thoroughly explored, then the functional changes to the engine API are presented. The book reads as though LaMothe is speaking directly to you while transcribing his thoughts to the page.

In the final section of the book, LaMothe tackles several advanced graphics topics: perspective texture mapping, spatial partitioning, shadows, and anima-

★★★★★	excellent
★★★★	very good
★★★	average
★★	disappointing
★	don't bother



tion. The visibility chapter is particularly strong with an in-depth look at Binary Space Partitions (BSP trees) among other various portal techniques. The engine code and examples are well commented making it easy to jump back and forth from the book to the source code.

The companion CD is as robust as the book. It contains a bevy of additional resources, including all the source code covered in the text (precompiled executables, appendices, 25 articles from various authors on everything from artificial intelligence to Pentium optimization, QUAKE source code, trial versions of some helpful game development tools, and the DirectX 9 SDK. The modeling tools are a very nice touch and add to the completeness of the overall text.

Simply put, this is a thoroughly satisfying book. While LaMothe's approach in developing the engine is sound, understand that he makes design choices throughout the book to make a fast software engine (no shaders, no complex light models, lookup tables). The theory behind his choice in approach is the valuable part of the book and the engine is a practical demonstration. Readers looking to develop their own engine or understand the behind-the-scenes details when using an API like DirectX will truly appreciate the effort LaMothe has undertaken.

★★★★ | *Tricks of the 3D Game Programming Gurus* | Sams Publishing
www.samspublishing.com

Jeremy Jessup works for Rockstar San Diego and has been in the game industry for over five years.

IDV's Speedtree RT 1.5

Speedtree RT 1.5 is a hybrid 3D modeler and run-time engine that helps developers integrate great-looking trees in real-time applications quickly. The focus is not so much on best-possible quality, but in keeping a balance between quality and real-time suitability. Tree handling is divided into two different subtasks. First, you generate realistic tree models using an editing tool, then you apply proprietary algorithms to render them efficiently. This approach is interesting and attacks the core issue in creating realistic trees quickly: mapping modeling to rendering.

The first component in the Speedtree RT suite is the tree modeler, which can be used either as a stand-alone application or integrated within 3DS Max. Speedtree creates hierarchical models of each tree, so leaves are linked to their branches, branches to the trunk, and so on. This means trees can be animated quite realistically, with leaves swinging and branches arching depending on the wind strength. Modeling is a hybrid of geometry for the trunk and fronds, and image-based, screen-aligned billboards for the leaf clusters. Because trees are modeled parametrically, you can adjust features such as branching, the amount of leaves, and so on, to help you create any new, unique species.

The second component is an API that handles all the tree loading, LOD handling, and rendering for you. Speedtree's run-time component interpolates using blending between several discrete LODs that are automatically computed by the modeling tool, and renders the whole thing as a billboard when the tree is very distant. The beauty of Speedtree's approach is that all the LOD handling, dynamic tree lighting, shadow computation, and animation are hidden away, so all you need to do is use a simple programming interface to access the tree technology.

To test Speedtree, we integrated it with an existing DirectX project already involving large outdoor scenarios. Speedtree comes with sample implementations for DirectX, OpenGL, and Netimmerse, so all we needed to do is copy and paste from the DirectX examples. Overall, it took one day of work of one developer to have our first trees working, and about three days to have a complete version, with trees integrated with the scene graph, animation, and collision detection. All in all, the API is simple and well laid out, so integration is quite straightforward.

The resulting trees are a perfect balance between speed and quality. Seen up-close, the trees are remarkably believable: you can look at a treetop while standing directly below, and the illusion of volume and parallax between the leaves is properly maintained. The use of pure billboards for distant LODs allows you to easily create large vistas with forests. Some individual trees do look a bit algorithmic and fractal at times, but when placed in a grove or forest, results are strikingly realistic: animated trees with real-time shadows that you can see far away and up close.

Compared with previous versions, Speedtree RT 1.5 adds frond support and better performance. The new trees have richer, denser branches, and their internal structure just looks more realistic than before. The new version can create compelling shrubs and bushes as well.

Providing more code samples would be a great way to expose all the potential to the user. Advanced topics such as shadows on uneven terrain, scene graph integration, fog, and seasonal changes are ideas that will definitely pop into your mind when using this package, and having an example at hand when you feel like coding advanced ideas would be fantastic. That said, the code examples cover most day-to-day uses sufficiently, from tree loading, rendering, and the setting of the different parameters such as wind, light, and so on.

Speedtree RT 1.5 can be purchased on a per-project basis, as most game development toolkits out there. The cost is $5,995 per title, which will make sense (or not) depending on your cost structure. Before balking at the price, consider how long it would take to develop a comparable technology internally at your company. Speedtree combines reasonable cost, short time-to-market, and strikingly good results.

★★★★ | Speedtree RT 1.5 | IDV Inc.
www.idvinc.com

Daniel Sanchez-Crespo is the founder of Novarama, an independent game studio in Barcelona, Spain, that creates games for the PC and Xbox platforms.

Return

Lorne's Oddysee

Oddworld's Lorne Lanning on the Art of Storytelling in Games

Oddworld's Lorne Lanning believes the message is the key.

Lorne Lanning spent two and a half years fine-tuning what would become Oddworld before being given the opportunity in 1997 to actually develop the series's first adventure, ODDWORLD: ABE'S ODDYSEE. Since then Lanning has kept giving players bigger and bigger glimpses of his own private world.

One of the characteristics that have set Lanning's games apart is his almost obsessive focus on the importance of storytelling; narration and character development are central to his games, with action skills receiving secondary attention.

With a renewed interest in the art of storytelling in game development, *Game Developer* visited with one of its apostles.

Game Developer. What are some of the bigger challenges in implementing good storytelling within a game?

Lorne Lanning: I think the primary obstacle is that games are built upon a limited chemistry of repeatable mechanics. Successful gameplay is the reasonable ramping of frequency and balance from these repeatable mechanics to create an additively progressing challenge. Great stories are built upon a very different structure. Stories do not repeat their subject matter; they continue to pull us forward via the pacing as dictated by a director and/or writer per the changing circumstances of an engaging character. By nature, the two mediums are in conflict. One is based upon repeated functions and the other is based upon continued change.

GD: So how does Oddworld try to implement these two sides of a game development coin into a cohesive playing experience?

LL: We try to create compelling heroes that are able to manage an interesting plight via the process of ongoing repeatable mechanics. For us, as the hero performs his repeatable actions, the actions need to be related to the narrative motivation and the character's development. The process of our mechanic creation is typically one that is directly distilled from who the character is and what his plight is. This means the mechanics need to be innovative. This aspect is critical if you want to create characters that people feel for, yet they feel for them while in the gaming experience.

GD: What key factors can be used to check a story line's effectiveness in a game?

LL: As far as universe development is concerned, a great interactive story must be built upon four critical components: unique characters, unique settings and environments, unique actions, and unique dilemmas. However, these components only form the soil from which you might grow an interesting character plight; you need compelling circumstances. Would this character's plight be interesting if it were not within a game's context? If not, then it's probably not going to be that interesting in a game either. The plot needs to stand on its own regardless of the medium. Intrigue and character development are medium-independent; we are emotional and intellectual beings. We want to be taken for a ride that engages our mind and stimulates our senses. If this can be achieved while also providing us with a challenging experience that stimulates our competitive or cooperative natures. . . then we might have a winner.

GD: When is style more important than substance?

LL: I don't think it ever is, though I suspect that at times some of our work has had more style than substance. When this has happened, it was the result of overly ambitious design that was beyond our realistic capabilities. You then fall into what I call "reactionary design." You're trying to find Band-aids for work efforts that didn't quite fully manifest, so you're left with a bunch of partial assets that need still need to deliver at a certain time and for a certain budget.

GD: What are some of the biggest changes in game development industry you've seen since ODDWORLD: ABE'S ODDYSEE first launched?

LL: Innovation in game design has become more difficult due to a publishing climate that is growing more afraid of creative risk-taking. You can't really blame them, as the stakes are getting higher as production costs increase while the number of retail winners continually decreases.

GD: How do you keep Oddworld fresh for ongoing fans while tantalizing for newcomers?

LL: I think it's important that you keep a certain consistency for the brand while hopefully surprising the audience; as soon as the audience thinks they've got your number, you're dead. Innovation is the key, yet innovation compounded atop a unique universe that you've already put out there and that's already been received well. You need to convince the audience that they aren't going to know exactly what to expect, except that it will be different and it will be the product of a team that really cared to deliver something special.

It's very difficult to achieve sustained interest if you don't deliver products frequently enough. This is something that we've always been trying to rise above.

Adaptive Compression Across Unreliable Networks

In a networked game system, we want to compress our network messages to reduce bandwidth usage. Last month ("Using an Arithmetic Coder: Part 2," September 2003) we compressed our output using an arithmetic coder with a static data model. The data model consisted of some statistics we generated by analyzing training data; the arithmetic coder then used those statistics to compress transmitted data.

This approach offers limited effectiveness. Often, our network messages will have characteristics significantly different from the training data; in these cases, compression will be poor. File-based arithmetic coders deal with this problem using adaptive compression: as data is processed, they modify the statistics to conform. Successful adaptive compression depends on the decoder's ability to duplicate exactly the changes enacted by the encoder.

Unfortunately, in high-performance networked games, we want to send most of our data in an unreliable manner (for example, using UDP datagrams). Because of this, the straightforward implementation of adaptive compression won't work. The server would base its statistics on messages transmitted, and the client would base its statistics on messages received. As soon as a single message is lost on the network, the client falls out of step with the server. The client is now permanently missing data needed to reproduce the statistics, so everything it attempts to decode in the future will be garbage.

One Possible Solution

We might solve this problem by making the server authoritative over the current data model. That is, both client and server start with the same statistics. As the server sends data to the client, the server measures the statistics but doesn't yet incorporate them into the data model it's using for transmission. Every once in a while, the server gathers together all the statistics from recent messages and builds a new data model, which it transmits to the client. As soon as the client acknowledges receipt of the new model, both client and server switch to this model for future transmissions.

This solution will work, but the fact that we need to transmit the statistics is a major drawback. One of the most attractive aspects of adaptive compression, in the domain of files, is that the statistics are implicit and thus take up no space in the compressed output. It's a shame to give up that advantage; if we do, suddenly adaptive compression becomes a questionable pursuit. For adaptive compression to be useful, we need to save a lot more bandwidth (through adaptation) than we spend (by transmitting the statistics). But generally, the more effective statistics will be, the more space they require. So really, we'd prefer a solution that allows the statistics to remain implicit.

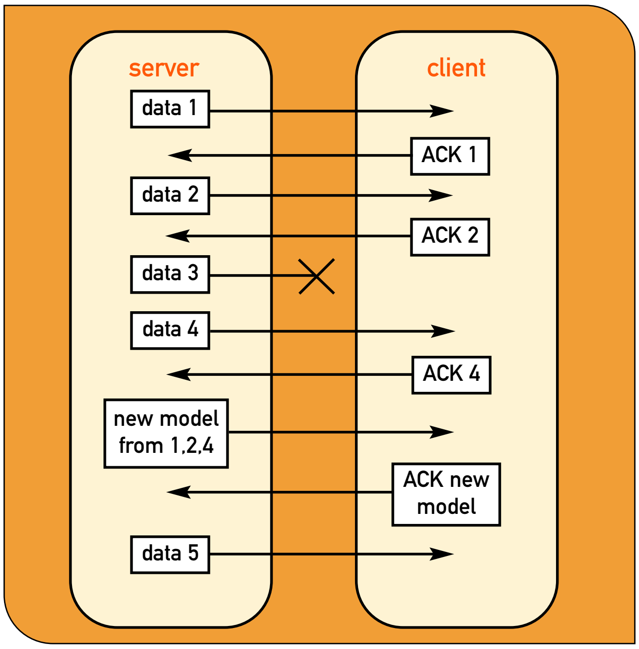
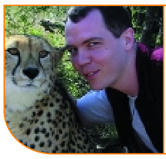


FIGURE 1. Cumbersome protocol for adaptive compression. Data packet 3 is lost on the network; since the server never receives an ACK for it, the server builds the new model only from packets 1, 2, and 4.

An Efficient Solution

Despite packet loss, there is enough shared knowledge between the client and the server to enable adaptive compression with implicit statistics. I'll start with a straightforward yet cumbersome method of accomplishing this; then I'll refine the method.



JONATHAN BLOW | *Jonathan is a game technology consultant living in Austin, Tex., although he may run for star of* Terminator 4. *Advise him at jon@number-none.com.*

Though the server can't predict which packets will actually reach the client, the client can tell the server this information after the fact. Suppose the client acknowledges each message it receives; the server then uses these acknowledgements to build a new data model. This new data model uses only statistics from messages the client received. The server must then tell the client when to start using the new model and which messages were used to build that model, because the client doesn't know which ACKs the server received (ACKs can be lost too!). In a naive implementation, the server and client need to stop and synchronize when switching the data model, because the server needs to make sure the client knows the correct statistics before continuing with new packets. Figure 1 illustrates this protocol.

So far there's a lot of round-tripping and some annoying synchronization. Fortunately, all that can be eliminated. First we perform a conceptual refactoring and make the client (or more generally, the message receiver) authoritative over the data model. Because the receiver knows which messages were not lost, it's in the best position to choose the model. Instead of ACKing packets individually, the client just tells the server, "Build a new data model, based on the old one, but including also the statistics of messages 0, 1, ... and 9." So far this "Build a new data model" message is nothing fancy, essentially just a big batched set of acknowledgements. In a moment, though, we'll make it a little more powerful.

This request for a new table might be lost or delayed, and we don't want to synchronize on such an event. So we enable the server and client to keep track of some small number of old data models. In the sample code (more on that later), I arbitrarily chose 7. Each packet is labeled with the index of the data model that was used to encode it. So suppose the client and server are both using model 3, and the client sends the request "Build model 4, starting with model 3 and mixing in this list of messages." If the request succeeds, the server starts sending packets labeled as model 4, and the client knows how to decode that (because the client built model 4 for itself at the same time it told the server to build model 4). If the request is lost on the network, the server's packets will continue to be labeled as model 3. Because the client still remembers model 3, it can decode these successfully. All this is illustrated in Figure 2.

This data model index is prefixed to each packet, so it requires a little bit of extra bandwidth, perhaps 3 bits. So that packets can be ACKed, we need to give them unique identifiers, which takes some more bandwidth. These amounts are small, though, compared to the savings we ought to get from compression. Bandwidth overhead is not a problem with this algorithm. It's with memory that we may see a problem.

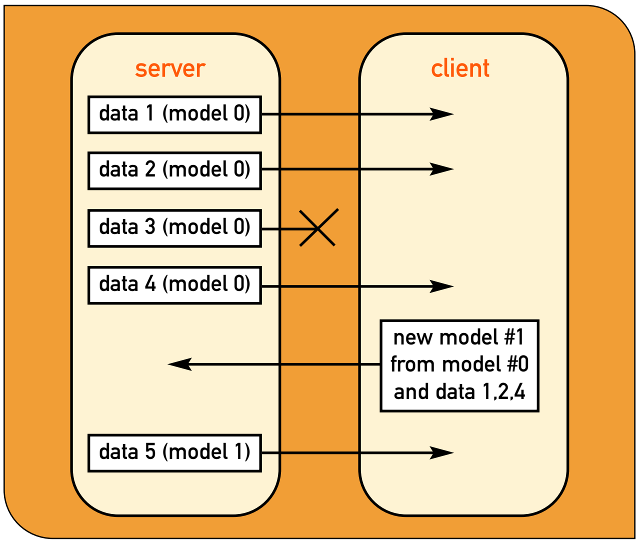


FIGURE 2. Streamlined protocol where the client controls the data model and synchronization is unnecessary.

Reducing Memory Usage

Because the server doesn't know in advance which messages will be used to generate a new table, it needs to remember statistics for each individual message it sends. The memory usage is not so bad if you are just thinking about transmitting to one client. But if you are making an MMO game with 1,000 clients logged into the same server, suddenly we're talking about some serious overhead. Exactly how much overhead depends on the data model, but keep in mind that data models can be quite large. The simple order-1 model from last month will often be tens of kilobytes, and you want to use something more sophisticated than that. Fortunately, though, incremental data model storage techniques may help you keep these sizes under control.

On the other hand, instead of storing a large high-order probability table for each message, the server could just store the messages themselves. When a request comes in to build a new model, the server can generate the statistics for each message by decoding them just as the client would. That adds to our CPU cost, though, since the server is eventually decoding just about every message it encodes. Decoding tends to be more expensive than encoding, so we will have more than doubled our CPU cost. The extra cost isn't expensive enough to worry about for a single stream, but in a system with 1,000 clients, the cost may be prohibitive.

But even storing messages like this, our memory usage may still be uncomfortably large. If we're transmitting 1 kilobyte per second to each client, and we update a client's data model every 60 seconds, we need at least 60 kilobytes per client; multiplied by 1,000 clients, that's 60 megabytes, which isn't funny. That minimum figure assumes no packet loss among any clients, which is unrealistic; and we still must pay the hefty CPU cost to decode all that. Clearly, we'd like a more attractive solution.

My approach is to group messages into batches, say, 10 messages per batch. The server remembers one data model per batch, and the client requests new models by specifying the batch numbers. (The batch number for each message is just its sequence number divided by 10, rounded down to yield an integer.)

This batching degrades the performance of the compressor slightly in the event of packet loss. If the client fails to receive some message from a batch, then none of the other messages in that batch can be used in any new data model (because the server has mixed the statistics from those messages all together and can't separate them again). Because of this the effect of packet loss is magnified. For a batch size of 10, 1 percent packet loss means 10 percent of batches are lost; 5 percent packet loss means 40 percent of batches are lost; 10 percent packet loss means 65 percent of batches are lost. These numbers sound high, but they're really not so bad. If the packet loss gets higher than 5 percent, the game is likely to feel unplayable anyway (due to general poor connection quality), so we're really only worried about losses below that rate. But as it happens, the adaptive modeler performs surprisingly well even with 10% packet loss. Apparently, we can still do a reasonable job of tracking message statistics even without seeing the majority of the messages. Table 1 shows the resulting compression rates for the sample code at varying rates of packet loss.

Forgetting Message Statistics

Once the server uses a batch of statistics to build a new data model, it can forget those statistics and free up memory. But to cope with packet loss, and to prevent denial-of-service attacks, the server will also need to discard unused statistics when they become old enough. This raises the possibility that the client will request a new model using batches the server has thrown away. That's not a big deal; the protocol handles it just fine. When the server finds that it no longer knows the statistics for one of the batches, it just ignores the request. The server continues onward, using the old data model for transmission. The result is the same as if the client's request were lost on the network.

The Sample Code

The results in Table 1 were measured from this month's sample code, which you can download from the *Game Developer* web site at www.gdmag.com. The code performs a simple simulation of a client/server system. To keep the code focused, data is not actually transmitted over a network. Instead, the program's main loop passes messages between objects representing the client and the server, with a random chance for each message to be destroyed instead of relayed.

Like last month, the code uses training data to build an initial data model. But this time, that model is used only as a starting point. Afterward, the data model is updated at the client's request. Because both client and server compute the same data models for each message batch, the data models stay in sync.

The server transmits a 300k stream to the client; this stream consists of the first two books of *Paradise Lost*, followed by the comp.graphics.algorithms FAQ, followed by some interesting C++ code. As you can see from Table 1, we achieve roughly 50 percent compression, which is pretty good.

Adjustable settings are provided for you to control packet loss, maximum message size, batch size, and the weight new message statistics are given when mixed with old ones; you can play with these values and see how the results change.

As I already mentioned, this compression algorithm requires unique identifiers (such as sequence numbers) on all unreliable packets. In addition to allowing compression, these sequence numbers can be put to other uses. For instance, we can use them on the client to estimate the current server-to-client packet loss; that estimate can be very useful in dynamically adjusting the behavior of the network protocol to maximize performance. The sample code measures the approximate packet loss and prints it out for you after the test is done.

The Future Code

The sample only allocates a fixed range for sequence numbers; if you try to send too long of a message, the sequence numbers will overflow, and the protocol will malfunction. For a real game you will want to change this. It's not hard to handle; you just need the client and server to understand that sequence numbers will, at some point, wrap back around to 0. However, I wanted the sample code to be minimal and clear, so I left this out.

Currently, the client and server only use one data model at a time. Though they remember a history of several models to avoid synchronization, only the most recent is used by the server to encode outgoing data. We might improve compression efficiency, at a large CPU cost, by having the server attempt to encode each outgoing packet using all of the consensual data models, transmitting the version that came out the smallest. I leave this as an exercise to the interested reader, since the CPU cost on current hardware would make this approach unattractive to many people.

TABLE 1: COMPRESSED DATA

Uncompressed size is 296,906 bytes, 10 messages per batch, 500 bytes per message. The line for 100% packet loss tells you how effective the static data model would be on its own.

Packet Loss	Compressed Size (bytes)	Compression
0%	148,209	50.1%
1%	149,863	49.5%
5%	154,407	48.0%
10%	159,352	46.3%
100%	185,890	37.4%

Return

15 Things All Artists Should Know: Part 2

Last month I presented seven points on my list of 15 things all artists should know, based on what I learned over my years in the game industry. To close out the series and my role as Artist's View columnist, here are the remaining eight.

Photo by Stacey Gadwill/Istock Photo

8. Go digital. I imagine that I am preaching to the converted here, but if at all possible get the best digital camera you can afford (even secondhand ones that are a few years old now are perfectly good) and weld it on a chain around your neck.

I cannot count the number of times over the past few years that I was driving somewhere with my family and passed the most perfect set of ruined farm buildings or concrete storage units and had to curse the fact that my camera was at home or not charged.

Unlike days gone by when the expensive and time-consuming process of getting a photo onto your computer made it less practical, digital cameras mean you can take pretty much as many pictures as you like and have them prepped as textures in no time flat, and at no extra cost.

In addition to the digital camera, it may also be advisable to assemble a Game Artist Photo Sourcing Apparatus Container (GAPSAC™), which would typically include:

• Bolt cutters, to get into those secret government compounds where all the best military vehicle textures are found.

• A lead-lined bodysuit and some thick rubber gloves for protection in the condemned uranium processing plant, as you search for those vital shots of corroded metal and rusting machinery.

• The phone number of a good law firm.

You don't qualify as a real game artist until you have been arrested for trespassing and spent the night tied to a chair in a disused warehouse being interrogated by an unofficial branch of the CIA.

7. To meet or not to meet? One of the great things about working as an artist in the game industry is that we get paid to spend our days creating four-headed zombie chinchillas or designing the interior of Commander Grurg's alien mothership. Talk to just about anyone working in an area like retail or manufacturing and ask them when was the last time that they had to spend a day deciding what a Tantric Horn Demon looks like and watch them shake their heads in feigned disgust while they secretly wish that they could swap places with you.

Our job is pretty cool (if you like this kind of stuff, of course) and much of what we do as artists is about as far away from a regular job as could be imagined. However, if we are to stand a chance of finishing a project on time and within budget, we need to plan our work in much the same way as any project-based industry would, and this pretty much always means meetings.

It's not that meetings are bad, on the contrary, they're cer-

HAYDEN DUVALL I *Hayden lives with his wife, Leah, and their four children, in Garland, Tex., where he works as an artist at 3D Realms. Contact Hayden at haydend@3drealms.com.*

tainly important, but sitting around a table planning work schedules or discussing design points can easily change into a debate about whether or not Batman could take Daredevil in a straight-up fist fight.

This is an unavoidable side-effect of the industry we are in and more importantly the type of people it attracts. It is very easy to get carried away when sitting and talking about the game you are making. The trick is to find that ideal balance between the meeting vacuum, where no one knows much about what's going on, and meeting overdrive, where the working week ends up being four days of meetings and one day writing up the outcome of those meetings.

6. Living on the edge. Working in a technology-driven industry, much importance is put on the value of being on the cutting edge. Over the years I can remember many games that were pushed on the strength of their "26 levels of parallax scrolling" or "full three-dimensional vector graphics" and the game press and publishers' marketing machines generally latch on to such advancements with vigor.

For the artist, more often than not advances are a good thing. The more tricks we have to dazzle the player with the better, but there is danger inherent in living so close to the edge.

Just because something is new, it doesn't actually mean that it is better. A game's visuals need to be evaluated in terms of the setting, atmosphere, and context of the game and how they can best represent the world and characters that are being created. Just because you are suddenly able to do dynamic volumetric explosions doesn't mean that you should cram them into every corner of your game, especially if it is set in a medieval castle. Cutting-edge reflection technology can easily push a game toward hordes of reflective surfaces, but this is probably going to look a little odd in a game that takes place in a giant forest.

Integrity of artistic vision is a luxury not generally afforded to artists in games, as so many external factors have to be taken into account, not the least of which is the need to incorporate new technology when possible, but it is all a question of balance. At the end of the day, a visual experience that allows the player to get the most out of your game is what it's all about. Being able to get the most out of your technology while keeping things coherent is a valuable skill to have.

5. Imagination is not universal. The next time you find yourself telling someone about an amazing dream you had the night before, watch carefully how they react. I guarantee you that any interest they are displaying is nothing more than a thin veneer, masking the profound boredom underneath. O.K., maybe that's a bit harsh, but it's very hard for someone else to truly re-create an imaginary experience from nothing more than your description.

This phenomenon carries over into the visual design process for a game, whether it is early on when an idea is being pitched to potential publishers or investors, or later in the game's development where designs are submitted for approval by the Powers That Be.

A valuable lesson that I have learned over the years is that if you assume that those to whom you are submitting your ideas have absolutely no imagination whatsoever, chances are you will have more success. A written description is bad, a sketch with a paragraph of accompanying text is barely acceptable, a full painting with additional drawings for support is beginning to knock on the door of reasonable. A selection of renders of a static 3D model is moderately satisfactory, an animation of this model if it is a character, or a fly-through if it is an environment is good, and a real-time, in-engine working demonstration is perfect. Other people's powers of imagination are unpredictable, so using as comprehensive a presentation as possible will be infinitely more reliable.

4. Everyone's an artist. Well, that's not strictly true, but everyone has an opinion on art. How many times have you heard a non-programmer look over a programmer's shoulder at a screen full of C++ and say, "Hmmm, I'm not sure I like those parentheses or that algorithm . . . maybe if they were in a different font"? As an artist, you've got to expect that everyone is going to express an opinion on your work and that chances are not everyone will like it. The thing to learn is that just because someone has an opinion, it doesn't mean that it matters.

Sure, that might sound a bit arrogant, but if I were to eat at any of the world's most exclusive restaurants, chances are that I wouldn't like the quail egg frittata with aromatic duck spleen pate. I could go and tell the chef, but I don't think he would give a tinker's damn what I thought, as my opinion was just a subjective expression of taste from one of many who would be eating his food. Now if I were the owner of the restaurant or a well-respected food critic, he may sit up and take notice, but otherwise, my opinion wouldn't make any impact at all.

You too have to develop confidence in your own judgment as an artist and learn to filter out many of the opinions that you will hear as you work. Listening to people is often helpful, but you will never be able to please everyone that wanders over to your monitor, so trusting your own ability is vital.

3. A world of filth. Figuring out what will work best visually is as much a part of an artist's job as is producing the assets themselves. I am probably not alone when I say that I have often had what I thought was a great idea, that in practice proved to be a horrible disaster. If our industry was stationary and we didn't have to constantly absorb the effects of new technology, this task would be easier, but as it stands, we need to reassess our visual boundaries continuously. There are, however, some reasonably robust ideas about what in general terms is easier to create successfully within a game world, as follows:

Dirty is easier than clean. Clean surfaces can often be boring on the screen, and while they are in fact simpler visually, it is this simplicity that makes them harder to re-create in an accurate, interesting way.

Old is easier than new. By extension of the idea preceding, creating an object or area that is supposed to be new is also more difficult than making things or places that are worn and weathered. Surfaces in a game world need to be more interesting than their real-world counterparts if they are to avoid the appearance of an empty or plain-looking environment.

The future and the past are often easier to re-create than a contemporary setting. We (and that includes the player) are less easily fooled by the things with which we are best acquainted. A walk cycle, facial expression, desk lamp, or shopping mall are things that we come into contact with all the time, so our subconscious ability to compare what we see on the screen with what we know from experience can easily highlight shortcomings.

One answer to this problem is to always work on games that are set completely in a fantasy world. But on a more practical level, it is best to use as much quality source material as possible. If you need to build a church, do some research and create something that's accurate. If your world needs elephants, spend some time traveling in Africa (you've got to at least try to get them to fund it), or failing that, look for a local zoo.

Designing a game in many ways is like riding a roller coaster, complete with ups, downs, and spins.

2. Mr. Cleese and his rotating knives. An obscure reference perhaps, but I feel confident that a good proportion of this column's readership is familiar with Monty Python. One famous Python sketch has John Cleese portraying an architect who is trying to win the contract to design an apartment block by making a presentation of his ideas in front of a selection committee. The only problem with this is that Cleese's character has previously only designed slaughterhouses. The presentation goes well until Cleese mentions the addition of rotating knives to the apartment block's foyer, and despite his attempts to defend his ideas showing how innovative and creative his designs are, he is ultimately rejected.

What has this got to do with making art for games? Well, in a rather odd way, it illustrates that as an artist/designer, your work has to be focused on the game you are working on, taking full account of both its style and context. Art that intrudes on gameplay, whether it is crushing framerate or making navigation difficult and confusing, or that simply conflicts with the overall aesthetic of the game, will be unsuccessful regardless of how good it is in isolation.

1. Ride the roller coaster. Making a game can rarely be characterized as a linear experience. Starting at point A with a comprehensive design, full team, and solid technology, progressing smoothly to completion at point B isn't a story you'll often hear echoing around the halls of GDC. Naturally, making the art for a game follows an equally convoluted path.

As an artist, you are called upon to produce your best work throughout the entire life cycle of the game you are working on. Being aware of the ups and downs that are likely to occur can help you fight the urge to beat yourself to death with a graphics tablet that is bound to arise from time to time.

At the start of a project, chances are that the art team will be mainly concerned with concept work. Almost by its definition, this kind of unfettered art is likely to look very nice indeed. Whether it's pure concept illustration or level and character tests, the concept phase is about finding the "look" and generating excitement. Feeling good about the project from an artistic standpoint at this stage is easy, and the opportunities ahead will appear plentiful.

However, once a project gets underway, it's entirely possible that early iterations of the technology you are using will severely limit the quality of art that can be created. Temporary art assets may be necessary and place-holder textures and animations will bring the visual quality crashing down. As production begins to ramp up, progress with the engine and refinements in the production pipeline may once again raise the standard of the art. Yet as the project reaches its final stages, visuals may have to be scaled back as content is now at a maximum, and frame rate is bound to take a hammering. Final optimizations may allow the art more room to breathe, and the last few tweaks not withstanding, the game ships and you move on.

This pattern is by no means universal, but it represents one example of the art roller coaster. Simply understanding that a similar journey is always likely to take place can help artists maintain their enthusiasm throughout the life of the project.

Being a better artist is all about learning.

Final Word

Being a better artist is all about learning. Beginning to think that you know everything that you need to is the perfect indicator that in fact you don't. Things change rapidly in our industry, and we need to make sure that we continue to change too.

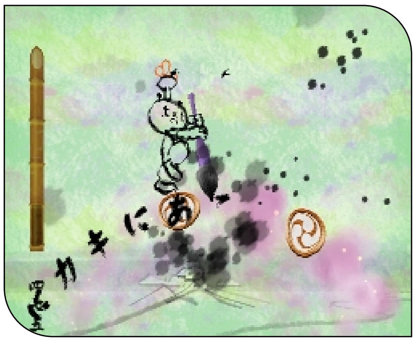
And finally, as I mentioned last month, I am handing over the reins of the Artist's View column. I would like to thank all those at *Game Developer* who over the last two years have contributed tirelessly to help me with this column. It has been a fantastic experience.

When the Music Is the Game

Since I started work on PARAPPA THE RAPPER in 1993, we have seen a decade of various *otogei*, or music-oriented games. The success of *otogei* in general is based on a couple of innate parallels between music and gaming. First, when playing musical instruments, one achieves a desired effect through motion. Furthermore, musicians who train and develop their sense of such motion improve their performance. The fundamental motions that create music are simple to perform: sing, beat, pluck, and press. Yet combinations of these simple motions can build considerable depth into the resulting work. These reflections of the gaming interface make it natural for us to integrate music into gameplay.

Stagnation in music games. There hasn't been a big *otogei* hit in Japan lately besides TAIKO NO TATSUJIN by Namco. Although the genre is experiencing a lull here, it will not be destroyed easily. Until now, Japan was a Mecca for lovers of music games, but recently we have seen the emergence of great titles developed in other countries such as AMPLITUDE, developed by Alex Rigopulos and the team at Harmonix. Innovative titles from U.S. and European developers help demonstrate that the music game genre is here to stay and not just a passing fad in game design.

MOJIBRIBON. The potential of music games is not limited to just rhythm action games. For example, our new title MOJIBRIBON gives players a more involving interface than just timing button presses correctly. The game uses voice synthesis technology to automatically convert text into rap sound. With this rap sound, players perform calligraphy on-screen. Some letters may appear patchy or blurred depending on the player's skill with the analog stick, and this affects one's score. The game rewards timing the brush strokes to the rhythm of the music with a volume of ink that can either be used or sent to another player through integrated e-mail software.


Blending rhythm and calligraphy in NanaOn-Sha's latest title MOJIBRIBON.

This style of gameplay may seem experimental, but one thing I discovered while developing MOJIBRIBON was that it is meaningless to develop a title if it is not innovative. This project faced a lot of difficulties in blending language and music, but the people at NanaOn-Sha and Sony Computer Entertainment gave me considerable support, for which I am thankful. Some also suggest that cultural issues may play a role, asking, "Won't it be difficult to release this title in the U.S. or Europe?" Yet it seems that the universality of music brings a degree of openness to music games. In fact, I receive a lot more positive messages, such as "I'm looking forward to the release of the title," from overseas than from Japan.

Jam session organization. One of the more interesting breakthroughs that came with developing music games was our organizational structure. Before the introduction of music games, a sound department was in a rather weak position relative to other departments such as programming or CG. Part of this was due to an institutionalized idea that those who create really great music make their livings by releasing CDs and playing on stage, not by composing music for videogames. As the music-game genre established legitimacy through its success, this misconception began to fade. As a musician, I prefer to do away with hierarchy anyway, and create a jam session–like atmosphere. It's an environment in which people can freely express their opinions, regardless of their position or responsibility. They can devote their energies and ideas freely to any aspect of the product they're working on, regardless of their specialty or title in the company. Granted, a flat organizational structure is relatively unique among Japanese companies, but giving everyone an equal voice made it possible for us to realize tremendous creative benefits.

Groovin' forward. One of the big changes we need to see in the music game arena is the introduction and proliferation of new input systems, to replace the ancient button-based input method. The old structure of having four buttons on the right is no longer the ideal interface for full utilization of high-performance hardware, and imagination is now far less restricted. Recently, in the area of arcade games, some developers found an easy way to diversify music games by focusing on building unique controllers such as turntables, guitars, drums, keyboards, and other musical instruments. Creative game developers should see this as an opportunity to experiment with innovative input systems too. I'm hoping that console manufacturers will focus their system development efforts in areas that can capitalize on such experiences, and push music game innovation forward.

MASAYA MATSUURA | *Masaya is the founder of NanaOn-Sha (www.nanon-sha.com) and the creator of music games such as* PARAPPA THE RAPPER, UM JAMMER LAMMY, *and* MOJIBRIBON. *He is delighted to be part of the videogame industry, although sometimes he wonders if he has any choice in the matter.*

Essential Game Gramm~~e~~ar

This month I'm stepping back from the usual meatier aspects of game design rules to consider of the foundation upon which those rules are built. I am not aiming to solve the proverbial chicken-and-egg question, but rather to examine some of the underlying questions of linguistics and human biology and how they may influence game rules, in the spirit of a growing movement to deconstruct game development.

Hatching a scheme. This column and The 400 Project in general represent an attempt to turn an analytical eye (but a practical one) on game design. Bernd Kreimeier has conducted roundtables at recent Game Developers Conferences — and has written several articles as well — attempting to survey and characterize some different approaches, including those of Doug Church, Chris Hecker, Chris Crawford, and others. Inspiration springs variously from architect Christopher Alexander's work on pattern languages, from rule- or lesson-based structures, and from software-engineering approaches.

Scrambling to avoid egg on one's face. It's tempting to ask which of these approaches is the "right" one, or even just which should be followed. But I think the field is too young for that. We still need to let all the embryonic theories come to maturity before we do any culling.

But we can still use inferences about the fundamental underpinnings of game design theories to help formulate them. I've personally found considerable inspiration from an unexpected source, in the works of the evolutionary biologist Steven Pinker. In particular, his book *How the Mind Works* (Penguin Books, 1998) is full of insights that I've found applicable to game design. Understanding the how and why of our brain's structure allows us to create more compelling and fun games. And some of Pinker's other books, notably *The Language Instinct* and *Words and Rules,* suggest some tantalizing ideas about what form more mature game design theories may take.

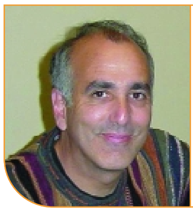
In CHICKEN RUN, the chicken comes first.

Bacon and EGGS. Pinker writes about how humans are born with instincts about language and grammar that are hard-wired into our brains, and how research also suggests that those brain structures may be evolutionary descendants of structures evolved to handle tool use.

Our associative brain structure that allows us to play such games as "Six Degrees of Kevin Bacon" or (as in my case) exhibit a perverse fondness for obscure puns, may in fact be thinking with mental "modules" specialized by evolution to handle tool use and language structure.Pinker theorizes that humans are born with our instincts about language and grammar hard-wired into our brains, and these brain structures may be evolutionary descendants of structures evolved to handle tool use. The associative brain structure that allows me to exhibit a perverse fondness for obscure puns may in fact be thinking with mental "modules" specialized by evolution to handle tool use and language structure. There certainly are many applicable analogies between the basic common concepts of nouns and verbs, the external and concrete examples of things in the world and actions that can be taken with them, and raw materials and the methods to shape or organize them to serve our needs. Alexander's very use of the term "a pattern language" implies as much.

I've found that using language structure as an analogy for game design is a powerful tool itself. One of my concerns about the use of Alexander's architectural approach to patterns is that they often feel too much like observations about existing things — nouns — without enough information about how to apply them, or how they can be acted upon — verbs — that I need to function as a game designer. Conversely, some essential insights about game structure, form, and desirable content are noun-like and not well-expressed as rules. Increasingly I am encouraged that these different approaches may converge into a unified whole, an Essential Game Grammar. This EGG from which all games grow may provide a lexicon of nouns and verbs, adjectives and adverbs, common phrases and syntax, and all the other elements that our brains are equipped to put into use. The 400 Project rules are purposefully expressed as active phrases, verbs to use on the nouns of game design elements.

Early text adventures used literal sentences like "use sword on thief" as user input while graphic adventures like THE SECRET OF MONKEY ISLAND combined written verbs with nouns in the form of images, letting the player click on "take" and then click on an image of a rubber chicken. Modern FPSes continue this evolution, where verbs are built into the actions the player can take: "shoot," "move," "strafe," and "duck." Next steps in this evolution are less obvious, but where there is an egg, there has to be a chicken, right?



NOAH FALSTEIN | *Noah is a 23-year veteran of the game industry. His web site, www.theinspiracy.com, has a description of The 400 Project, the basis for these columns. Also at that site is a list of the game design rules collected so far, and tips on how to use them. You can e-mail Noah at noah@theinspiracy.com.*

Return

rob bridgett & wolfgang hamann

Audio Department Structure and the Creative Process

Structuring Creativity

Illustration by Claudia Newell

Structuring how audio departments function within a development company is a unique problem. The lack of an industrywide standard within audio departments leaves greater scope for varied approaches and creativity, however it also allows for a great deal of time and money to be misspent. While audio departments are beginning to take on internal work structures similar to music, film, and television post-production environments, there is still room for a free experimental approach within our industry. With proper audio department workflow and personnel structure, it is possible to maximize creativity and support that creativity with a solid audio resource infrastructure.

Two Current Models

In the game industry there are two main ways in which internal sound departments can be structured. One way is to have dedicated sound personnel attached to a specific team just like programmers and artists. Here the aim is to have sound personnel sitting in the same team area as the artists and programmers with the notion that proximity increases communication, workflow, and creativity. Since sound personnel require soundproofed environments, edit suites are built in each game team area. However, for sound personnel to do their work at Fletcher-Munsen curve levels, doors must be closed the majority of the time, reducing or negating many of the communication, workflow, or creative advantages that proximity may have brought.

Since most of the work sound teams do occurs after there is enough art and code implemented on the target platform to characterize the game environment, real sound design gets pushed toward the end of the development cycle. During this period there is little time for creativity, since the game is developing rapidly and everyone wants to hear what the soundtrack will sound like. The push is on to get content into the game quickly.

Another model is based around the global sound team approach, where all game teams have support from a central group. Typically a sound lead or sound director is assigned to a specific game team for the period of time where sound is required. This may be for a three-week period during preproduction to create various iterations of sound design documents, and then again during the last third of the development cycle to create and implement the required content. In this model, the sound director is shared across two or more game teams, leaving little if any downtime for R&D.

The first model with dedicated sound personnel seems to provide the best avenue for developing creativity, since there definitely will be time while the sound team is waiting for the game design to settle down. But from the perspective of business efficiency, unless this downtime is kept to an appropriate amount, there is considerable waste. Who wants to sit around waiting for the game team to figure out an appropriate direction?

The second model of a global sound team appears to be much more business-efficient, but if everyone is constantly developing and implementing content, constantly moving from team to team, sound personnel may be too busy to be creative.

The Solution for the Global Model

The answer for the global model lies in specialization. Sound leads or directors must act more like record producers, directing a small team of sound specialists with their own contributions limited to appropriate areas of specialization. These leads can then supervise a small central team of specialists such as sound effects designers, composers, FMV post specialists, recording and mix engineers, dialogue editors, music and dialogue mastering engineers, and so on. A manager of the sound team would be there to guide and direct the team, as well as interact with the game team producers and senior management.

This model not only allows everyone to lead to their strengths, but also builds adequate time into the schedule to allow for creativity or R&D for each specialist as needed. Sound improves on an ongoing basis as everyone is constantly developing their own craft in a team environment with the appropriate amount of creative R&D time.

ROB BRIDGETT | *Rob is currently a sound director at Radical Entertainment, having previously worked at Climax Studios in the United Kingdom, where he was responsible for sound on a number of games including SUDEKI, SERIOUS SAM, and most recently WARHAMMER ONLINE. Rob attained his bachelor's degree in film and television at the University of Derby and later was one of the first to complete the M.A. in sound design for the moving image at the University of Bournemouth.*

WOLFGANG HAMANN | *Wolfgang is currently a game team project manager and manager of the sound department at Radical Entertainment. He also teaches a course called BMG: Business Management for Games at AI Center for Digital Imaging and Sound in Vancouver.*

The global model's flexibility also helps reduce unrealistic crunch periods when the art is late and the code is not quite ready or is too unstable to allow the sound artists to do their work properly. Resources can be immediately shifted to those game teams that need it most.

A global sound team also provides the opportunity for a group of highly skilled sound artists to all learn and grow together. In talking to those developers who had dedicated sound personnel on their game teams, the one common theme was that separating sound artists around different parts of the building greatly reduced idea sharing and creative growth. Separation also increased the amount of duplication in such areas as sound effects databases, third-party database software, and other tools. In addition, I found an unhealthy competition can arise around hardware and software resources, where everyone tried to keep up with the Joneses, even though much of this equipment wasn't really needed.

In the global model everyone shares a central sound effects database. Equipment is lent as needed, and since everyone is in the same area of the building, any workflow, software, and hardware issues can quickly be discussed and resolved. New processes and ideas are quickly and easily shared, providing an excellent training environment for new sound personnel. The sound director reports creatively to the game team producer and to the manager or director of the sound team for personal and professional development. In short, a true team environment develops, in which creativity can be built into any schedule for the benefit of the company as a whole.

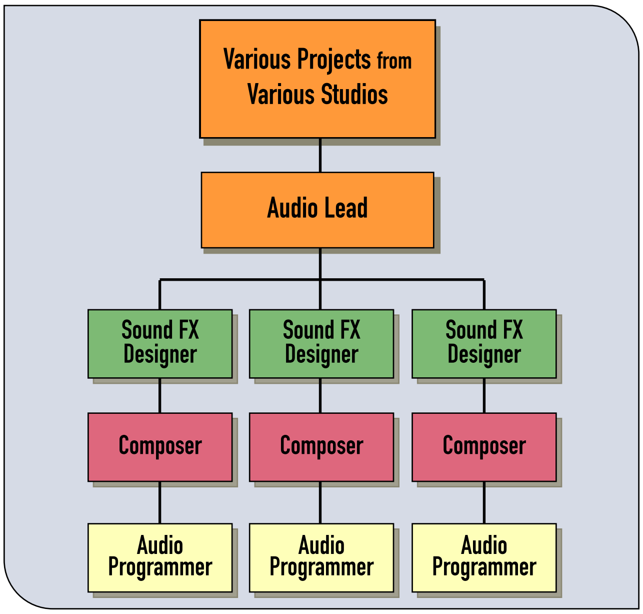


FIGURE 1. Organizational structure at a large company with centralized, non-project-specific resources.

Structuring Creativity

While structuring staff and department schedules to reduce unproductive amounts of downtime, it can't and shouldn't be eliminated entirely. It's important to and reconceptualize that time around maximizing its R&D benefit. Here we'll use an example of how downtime can contribute toward sound effects design.

Production downtime really should be seen as periods of experimentation and exploratory creativity. Distance from active production is an optimum time in which to experiment and to create new sound banks and effects relevant to the project or to future projects.

By its very nature, creating new effects will feed the requirements for specific sound effects further down the production line. You may see an animation of some strange creature or GUI feature and you instinctively know from your previous experimentations exactly what would work with the imagery you are seeing.

This type of work can also provide a refreshing break from the intensity of working on one project in one style. It's good to get away and work on sounds or music from a completely different and fresh style at least once a month. This flexibility increases morale and refreshes designers' creativity when they eventually get back fully to their core project. Working on one project only is intense and tiring, especially when a style is already aesthetically defined, such as the horror genre. Under this system, designers will tend to work intensely on one project and also, without realizing it, produce huge libraries of sound effects in other genres just to provide some temporary relief from the horror genre.

Usually building up large banks of similar sounds occurs in several defined categories, for example you might create a huge batch of sci-fi bleeps through a single design technique, and then a batch of squelchy horror effects using another. It's essential to categorize at this stage and keep these effects somewhere where you or other sound designers can gain rapid access to them.

"Genre relief" as a preproduction strategy is an extremely effective way of managing what would previously be called "downtime" in a schedule. Leveraged over several staff members, this new preproduction time will soon begin to create a large library of highly usable sounds.

This new sound effects resource can also help to alleviate the buildup of work that occurs at the end of the project. By having a new and pre-edited resource of unique sound effects, last-minute sound requirements can often be filled by sounds created earlier that are reworked or mixed with other elements into exactly what is required.

The concept also easily migrates to other areas such as music production, offering a great opportunity to experiment with new musical styles and create new and useful loops or ambient beds.

This creative use of downtime is essential within any audio

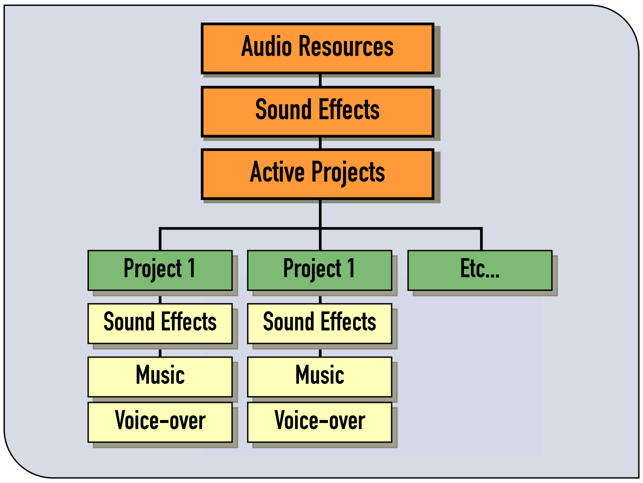


FIGURE 2. Organizational structure at a large company with centralized, non-project-specific resources.

production facility that has peaks and troughs of work patterns. Not only does it keep the audio staff fresh and fulfilled, but it also allows for all-important areas of experimentation. These experimental periods provide great ways in which to forge ahead into new areas of sound and music design.

The game industry is one of the rare institutions where a fixed musical style or structure to development simply does not exist. For almost every game released there is a unique approach to both implementation and style. By continuing to allow freedom within this area, we provide an excellent way of ensuring that game audio will constantly be moving forward both technically and aesthetically.

Getting Creative with Creativity

While providing more time for creative exploration is certainly beneficial, time alone can't raise the bar on innovation and skills development. We've seen some ways that sound artists can use downtime to great advantage on their own, but what about developing a higher standard of creativity and innovation through group activities?

Reviews. One way is to have monthly sound team creative or quality reviews, where each person's current work is presented to the team for constructive feedback. This may be the presentation of a current level, portion thereof, or FMV soundtrack. Be it sound effects, music, ambient effects, or anything, emphasis here is on the constructive. We all know there is a minimum of two options for any creative piece of work, and they are usually both right. However, when the team provides different options, the responsible sound artist may be provided with new, alternative perspectives. All ideas should be welcomed with no regard to their being right, wrong, better, or worse. Even the silliest of ideas can instill brilliance when bounced off of a welcoming creative partner. For this process to be effective, the phrase "That won't work" should never be uttered.

For example, hold a quality review occurring on the last Friday afternoon of each month, preferably after a milestone. Depending on the size of your team, a good four hours should be devoted to this exercise. The quality review is also a good team-building opportunity, as everyone can head to the local pub afterward to further discuss what was seen and heard.

To generate effective constructive criticism, there are a number of options available besides the usual brainstorming techniques most people tend toward. Everyone must throw in ideas, disregarding any sense of value, right or wrong. The initial goal is quantity of ideas, which have a better chance of hitting on something really brilliant. Everyone is encouraged to combine ideas to develop newer and more innovative concepts based on someone else's contribution. Criticism is banished entirely.

While brainstorming has some usefulness, it is not the only way to generate innovation and creativity.

Analogies. Another technique is to use analogies. In order to solve a given problem, such as a weak FMV soundtrack, a word is chosen from a preexisting list of randomly chosen words. Then, someone on the team chooses a random number. This number is counted down a list of random words according to the random number to arrive at a word. The word is then written on a whiteboard as a starting point, and everyone is encouraged to come up with additional words that are related to the original word chosen.

For example, our random keyword might be "refrigerator." What aspects of this word can we apply to our sonic problem with our FMV? Some words that "refrigerator" evokes might be: cold, white, food, icebox, vegetables, green, Freon, beer, mold, ice cubes, lunch, and so on. Now the group tries to apply each word to the problem scenario. The word "cold" might reveal that many of the sound effects have too much of the same upper-mid-frequency content and no bottom end. "White" might evoke thoughts of too much space in the arrangement, "food" might further elicit "balanced diet" and thereby reveal a lack of balance in the soundtrack mix.

Such an exercise may feel foreign and perhaps even silly, but it does actually work. Just remember to persevere. Eventually you will develop positive momentum. Analogies also seem to work best in smaller groups of three or four, so break up large groups if need be. Again, there is no such thing as just one right answer. The whole concept is to encourage and harness new and different ideas that come to mind.

Reversal. Another method that the group could use is reversal. Taking our FMV soundtrack for example, instead of saying it's too weak (even though we know it is), ask instead how it could be made even weaker. We could thin the mix by chopping top and bottom frequencies out, we could overcompress it, downsample it, replace various individual sound effects with

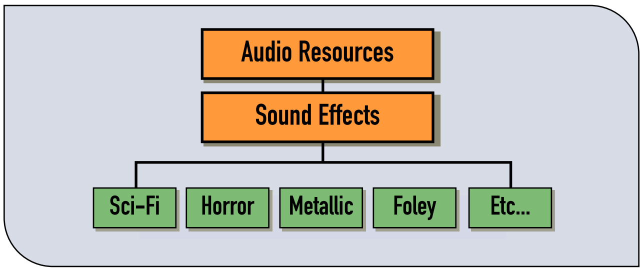


FIGURE 3. Categorizing in-house, non-project-specific sound effects.

even smaller-sounding ones, replace the real string quartet with a cheesy synth patch, let the company CFO loose on the soundtrack, and so on. As the process continues, the latter ideas become more and more creative until we stop laughing and realize that we've just come up with something quite novel and creative.

Aleatoric sound design. Another technique for inspiring creativity is to use chance. This can form another very entertaining yet seriously creative part of a group brainstorming session. It's something we call aleatoric sound design.

Chance plays a crucial part in any creative process — an idea can occur at the strangest and most inconvenient times. A good example is when we awake from a dream and have an incredible, fully realized musical score in our heads, yet as the daylight pours into the room the melodies and timbres slip away, never to be heard again. Perhaps later we will hear or see something, completely by chance, which reminds us of that music or the feel of the dream, and we are able to get some of it back into our consciousness.

Being able to exploit this creative technique, which can help to enhance a section of sound or music design that lacks punch or direction, is another way of approaching the design brief from a tangent.

In adopting the aleatoric process, we might layer in sound effects to an FMV, for example, at random. This often produces very strange and unexpected results. Animal sounds or screams under car or spaceship fly-bys is a great example of how to make your sound design stand out from everyday library sound effects which have already been used on hundreds of other productions. In addition, it often adds a layer of subconscious meaning to the soundtrack by giving the car sound effect a particular personality, which can become a narrative element throughout the game.

The same is true of music. If you try out several completely random soundtracks under our problem FMV, especially tracks that would normally be considered stylistically "wrong" for the project, there is a point where the music and image come together completely by chance. It's then a matter of identifying these new elements that do work and incorporating them into the final design.

These are a few possible alternatives to brainstorming that small groups can use to solve a wide array of problems. Many others can be used individually and in combination. An online search using the keywords "innovation techniques" yielded more than 20 books on the subject. The goal is simply to generate innovative solutions to sonic problems in the context of a highly creative and positive group dynamic.

Audio Resource Structure: The Foundations of Creativity

Structuring and creating new sounds for use on many different projects also raises issues of structuring the audio resources themselves. It's essential to have an ordered and highly logical resource infrastructure in place which everyone can access quickly to use in their own unique ways. The following section will take a look at how music and sound effects raw materials can be structured not only on a local level within the same company network, but also within satellite or smaller project studios.

Most larger development companies have the advantage of centralized sound and music teams in one place. These audio departments take care of production for all the projects currently ongoing within that company and are structured with resources readily available to all via local networks. Each project has at least one sound effects designer and one composer, overseen by an audio lead (or perhaps a team of specialist audio directors), or director who determines the overall aesthetic, implementation techniques, and scheduling for the sound and music production for all current projects. As a model this works very well for companies who have a centralized base of operations, illustrated very simply in Figure 1.

However, due to the satellite structure that some independent game developers maintain, these resources and even communication between audio personnel are all left to evolve separately as though they were different companies. So much could be learned from each other's previous experiments and experience that time and money can be saved and better products produced if the resources were effectively shared. Communication between audio personnel concerning audio design techniques and other areas of mutual interest could also greatly be enhanced.

Satellite Studios: Sharing Audio Resources

It's not uncommon for startup studios to use sounds on projects that are taken from the Internet or copied CDs (contractors may also be doing this, but there is no real way of checking). Illegitimate sound files pose a risk of copyright infringement issues should these files appear in a publicly released title — Particularly damaging, since the money involved in a lawsuit could potentially nullify any earnings for the development studio on a commercially successful title. Sharing the licensed audio resources freely between any number of satellite studios is a way of circumventing this problem while increasing the amount of material available to each studio.

All sound effects and music should be thought of as a group resource. This can be achieved via a very good, remote private network connection or via an audio resource FTP site. For example each project will produce a variety of reusable spot effects (also good for reference on potential sequel projects) that can be structured as shown in Figure 2.

And when creating sound effects entries from effects that are created in-house, again a simple easy to navigate structure is essential. One such structure is shown in Figure 3.

Maintenance and upkeep of a shared resource can be either done by one person or democratically added to by each studio themselves, comprising sound effects, music, voice demos, and documentation. A mechanism (usually an audio manager) should be in place to ensure that everything used is a legitimate licensed sample, and that things don't get out of hand and confusing in terms of directory structure.

Under this system, the unique satellite structure of some studios can be maintained, yet the managed resources will feed each group. Figure 4 shows how each group is fed by, and in turn feeds, a centralized resource (a local copy of which is kept on the hard disk at each studio location, meaning they only need to download and upload updates to the resource). This is also imperative as a backup resource.

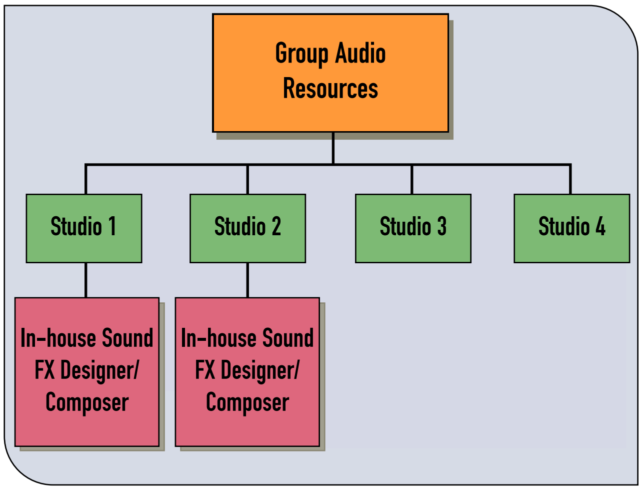

Group Audio Resources
Studio 1
Studio 2
Studio 3
Studio 4
In-house Sound FX Designer/ Composer
In-house Sound FX Designer/ Composer

FIGURE 4. Organizational and resource structure among satellites. Note that for the purposes of the illustration, Studios 3 and 4 have no internal audio staff but will be potentially brought online in the future.

Satellite Studios: Increased Communication

One of the main concerns over resource sharing among satellites is that using up to four or more offices, all potentially working separately on different titles, the audio staff somehow feel fractured and too much like separate companies each coming up with their own resource and creative solutions. Along with the sharing of group audio resources, such as sound effects, the feeling of community and collaboration among the separate sound staff at each office should be positively encouraged.

Each audio designer has his or her own specialties and styles. Information about each audio designer, in addition to MP3 demos of their work, can be made available to producers and designers in a database or via an internal web site. This information is useful for future project pitches as well as deciding to whom particular audio design tasks would best be allocated. Also, when there is a situation of too much work, other audio designers may be able to help with tasks such as voice file editing in order to meet milestones, for example.

A message board or similar resource on which staff can converse is of great benefit for sharing techniques and software tips. For example, global newsgroups could be set up on topics such as audio design, audio code, audio styles, audio reference, and so on. All staff can also share ideas and post questions and example sounds on the message boards, which will help the design of audio for all products on all levels.

Also of great use is a group meeting between audio staff as often as possible (once or twice a year if studios are very disparate), hosted by a different studio each time. Not only are such audio summits helpful and sociable, they also enable staff to suggest items for discussion, contribute presentations of work in progress, and in general strengthen the sense of community between the satellite audio staff. This method in effect establishes the global model over the large geographical boundaries enforced by the satellite structure of some development companies.

Finding Balance

Leaning on the internal human-resource structures of the post-production industries is a good foundation for thinking about managing and structuring creativity, but the game industry has its own unique workflow which determines the very nature of how the game audio department functions.

Exploring the ways in which the game audio industry functions differently from its sister industries can open up new avenues of creativity. However, the processes of encouraging creativity must be built on the solid foundations of structured resources, personnel structure, and communication. Finding an ideal combination of business efficiency and creativity is the key to determining the ideal approach for each individual studio, and adapting the balance of these resources to the changing demands of the work is an exciting and rewarding challenge for anyone lucky enough to be working in games.

Return

Taming Vertex Data Using C++ Templates

Problem: D3D low-level support. For manipulating vertex buffers, core Direct3D (D3D) only provides low-level memory management. Vertex buffer data is accessed using `void*` pointers. This level of support is quite correct in core D3D. D3DX provides higher-level support but only as a whole-mesh class. There is nothing between these high and low extremes.

In order to use a low-level vertex buffer (VB), you need to first create a vertex data structure, then create a VB of matching size, fill it with data (casting the `void*` pointer), and finally create a matching vertex declaration.

The application code must be self-consistent, with no help from D3D, and several low-level errors are possible. At best, errors will be caught by the debug version of the D3D at run time. At worst, they will produce application crashes. Often they are difficult to debug.

Problem: shaders bring flexibility. Prior to DirectX 8 (DX8), the fixed function pipeline (FFP) rigidly constrained the possible types of vertices. They were limited to a few combinations of standard elements: position, normal, color, and so on. Typical FFP games used only a handful of vertex types.

DX8 introduced programmable pipelines, specifically to address the rigidity of the FFP model. Programmable pipelines impose almost no structure on a vertex. A vertex only has to match the inputs to the corresponding shader. The standard FFP elements can be ignored, implied, or compressed, as described in the book *Shader X* (see References). Programmers can include nonstandard custom elements such as physics parameters, morph deltas, indices and weights, or partially complete lighting calculations. Useful vertex types are limited only by the ingenuity of the shader writer.

Core D3D provides only low-level, potentially error-prone support for vertex data. Yet, DX8 and DirectX 9 (DX9) positively encourage developers to experiment and create new vertex types. Low-level errors are even more likely when vertex data is malleable and vertex types multiply freely.

This article shows how to create a small template library for manipulating vertex data. C++ templates are a language feature that I feel are underused in the area of Direct3D programming. Here they structure the data, add type safety, and robustly automate some of the repetitive nuts-and-bolts tasks. Ideas from template metaprogramming provide flexibility to match the flexibility of shaders.

Assembling a Class from Components

C++ template metaprogramming (see Alexandrescu's *Modern C++ Design* in References) shows how a class can

IAIN CANTLAY | *Iain has been working professionally with 3D graphics since 1991, first with airline flight simulators (his employers included Evans & Sutherland) and later, beginning in 1996, with videogames. His credits include MACHINES and ART OF MAGIC. Currently Iain is the senior 3D engine programmer on WARHAMMER ONLINE.*

be composed from a list of elements at compile-time. Recursive multiple inheritance can be used to create a Composite vertex class from a list of Components.

```
struct EndList     {};
struct Position    { float x, y, z; };
struct TexCoord    { float u, v; };
struct Color       { DWORD c; };

template <class T1, class T2 = EndList>
class Composer : public T1, public T2
{
};

typedef Composer< Position,
        Composer< TexCoord,
        Composer< Color > > >
    TestVertex;
```

The typedef effectively creates a class, TestVertex, that inherits the attributes of Position, TexCoord, and Color. Figure 1 shows the inheritance hierarchy. Inheritance is not used in the usual object-oriented way. There is no polymorphism. The base classes are not used as interfaces. The inheritance is simply used to aggregate a list of component elements. Note how the empty EndList class serves as an end marker to stop the recursion.

The definition of TestVertex looks superficially like one of Alexandrescu's compile-time typelists. However, it is not. There is no need for compile-time manipulation of the list. In games, the types of vertices are typically well known in advance. Moreover, in DX8 and DX9, they have to match the inputs to the vertex shader. So, the types are predetermined by constraints outside the C++ compiler, and true metaprogramming is not called for. (The Composer class is inspired by Alexandrescu, but it is not nearly so clever.)

Other template syntaxes are possible. There are still compilers in use that do not support the default template parameters. In that case, two composer classes are necessary. The first takes two template parameters; the second takes a single parameter and acts, like EndList, to stop the recursion.

The two-class syntax would also be necessary if EndList added anything to the size of the composite class. On my compiler, it contributes zero size, but that is not guaranteed by the C++ standard (see section 1.8.5 of the ISO C++ Standard, Programming Language C++, in References).

The template syntax is unwieldy, even ugly. Alexandrescu uses macros to linearize his typelists. Applied to the example above, a macro would yield something like:

```
typedef COMPOSITE_VERTEX(Position, TexCoord, Color) TestVertex;
```

The macro syntax is certainly less cluttered. However, I use the template syntax. The recursive multiple inheritance is an odd new idiom. I prefer to make it obvious rather than disguise it. The choice is purely a matter of preferred style.

Recursive multiple inheritance is an obfuscated way to add a few member variables to a class, which is not useful as it stands. However, it becomes useful if you add more to the vertex components.

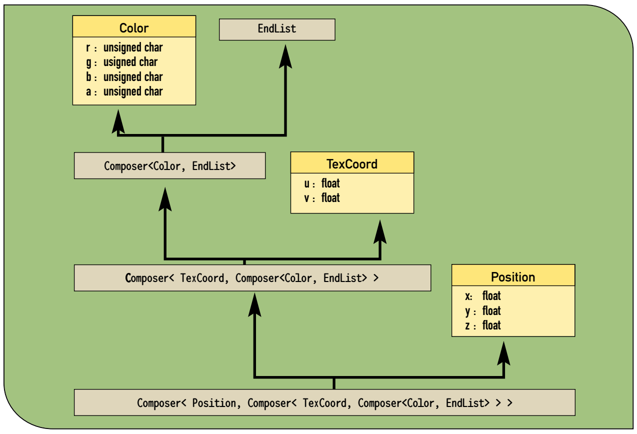


FIGURE 1. The inheritance hierarchy of a three-component vertex.

Building a Vertex Description

Give each vertex component an object that describes it:

```
class ComponentDescription
{
public:
    const std::string& name() const;
    size_t size() const;
    BYTE semantic() const;
};

struct Position
{
    float x_,y_,z_;
    static const ComponentDescription& description();
};
```

ComponentDescription objects are meta data — objects that describe other classes. The Composer classes can assemble ComponentDescription objects into a description for the whole composite vertex. Each Composer class is given a method, pushComponentDescription. Listing 1 shows how it recursively appends its corresponding description object to a list. (For now, assume that VertexDescription is a simple container with std::vector::push_back semantics.) Each method refers to its base classes assuming that one is an atomic component and that the other is another Composer class. Again, the EndList class is used to stop the recursion. Syntactically, it is interchangeable with a Composer, but its recursive methods do nothing.

The key technique of the whole system is that templates and recursive inheritance generate a cascade of recursive functions. The recursive functions build a description of their class and it is automatically guaranteed to match the structure of the class. Once you understand the use of recursion, the code is deceptively simple.

A cascade of recursive functions might appear inefficient for a real-time game application. However, the if statement in the vertexDescription method ensures that the recursion is only ever called once per vertex. Besides, the method is called infrequently relative to other time-critical tasks such as creating vertex buffers or setting state.

There are also extra unused vertexDescription methods. One is defined for every level of the recursive inheritance. The most-derived one hides all the others according to the rules of inheritance. They could lead to code bloat, but a good compiler should not instantiate them.

The recursive template calls are written once, in the framework library. They are not typically visible to users of vertices. By merely defining a vertex using the template syntax, the vertex description is automatically made available and now ready for useful work.

LISTING 1. The templates and recursive multiple inheritance create a chain of recursive functions that generate a vertex description.

```
class ListEnd
{
protected:
    static void pushComponentDescription(VertexDescription*)
    {
    }
};

template <class C, class V = ListEnd>
class Composer : public C, public V
{
public:
    static const VertexDescription& vertexDescription()
    {
        static VertexDescription d;
        if (d.nComponents() == 0)
        Composer<C,V>::pushComponentDescription(&d);

        return d;
    }

protected:
    static
    void pushComponentDescription(VertexDescription* pV)
    {
        // Push the type of the component that we add.
        pV->pushComponent(&C::description());

        // Recursively descend through the base class fns.
        // ListEnd implements nothing to end recursion.
        V::pushComponentDescription(pV);
    }
};
```

Uses: Creating a D3D Vertex Declaration

The VertexDescription is homomorphic with a D3D vertex declaration (see D3D documentation cited in References). Traversing the description can create a vertex declaration. Listing 2 shows that it is quite trivial to copy data from the ComponentDescriptions to the corresponding D3D structure and then create an IDirect3DVertexDeclaration9.

This method is written once in the framework and automatically provided for users of the framework. There is no scope for error because the D3D vertex declaration is automatically guaranteed to match the vertex data structure.

If FFP compatibility is an issue, the VertexDescription could similarly generate an FVF (Flexible Vertex Format) code.

The VertexDescription is almost identical to the D3D equivalent D3DVERTEXELEMENT9. So why create a different class? The D3D stucture contains information that is more dynamic, for two reasons. First, the Offset member depends upon a component's location within the vertex. The same component often appears at different locations in several different composite vertices. So it does not belong with a description of an isolated component.

Second, the Stream member indicates which stream a vertex buffer is bound to at runtime. The stream number changes as VBs are dynamically combined using the multiple stream features of D3D. A full treatment of multiple streams is beyond the scope of this article, but it is possible — the WARHAMMER ONLINE engine supports multiple streams. Streams are dynamically added and removed to support different levels of detail, and matching vertex declarations are automatically generated. More detail is shown in the sample source code (details provided at the end of the article).

Uses: Integration with a Vertex Buffer Wrapper

It is common practice to wrap D3D vertex buffers in a class. The VB wrapper shown in Listing 3 takes a VertexDescription as a constructor parameter. The constructor can take the vertex size from the description and use it to create a VB of the correct size. The VertexDescription is also saved for later reference. It is used later to verify that the correct vertex type is used when accessing the vertex buffer.

The VB wrapper class could be a template. It would be natural to parameterize it over the vertex type. The vertex size could then be acquired directly using sizeof. However, a non-template class is preferred for several reasons. First, most VB functions can be written without requiring the type of the vertex. Only the size of the vertex is required. Using the VertexDescription class provides sufficient robustness. Another reason is that if all the methods were templates, they would create unnecessary code bloat. Third, type-safe operation is only necessary when accessing the vertex data. Reading and writing vertex data is handled by a separate nested Access class. The Access class is a template, giving the type-safety where it is needed. For more detail, see the sample source code.

The final reason is that a D3D vertex buffer can contain more than one vertex type. Making the vertex buffer class a template would rule this out.

A useful VB wrapper class contains many more features. A full implementation is shown in the sample source code.

Uses: Checking Self-Consistency

The framework code has built-in self verification. Listing 4 shows the implementation in the method packingCheck. There are two types of verification: compile time and run time.

Run-time asserts verify that the component sizes (measured with sizeof) match the meta data. These checks detect user errors when adding new vertex components. For example, a class that inadvertently contains a virtual function will cause an assertion because the compiler adds a hidden virtual table pointer.

LISTING 2. It is trivial to traverse a VertexDescription and build the equivalent D3D data structure.

```
void ComponentDescription::setVertexElement(
D3DVERTEXELEMENT9& dst, BYTE off) const
{
  dst.Stream = 0;
  dst.Offset = off;
  dst.Type = type();
  dst.Method = D3DDECLMETHOD_DEFAULT;
  dst.Usage = semantic();
  dst.UsageIndex = 0;
}

IDirect3DVertexDeclaration9*
VertexDescription::createDecl(IDirect3DDevice9* pDev) const
{
  std::vector<D3DVERTEXELEMENT9> els;

  BYTE off = 0;
  for (size_t i=0; i!=nComponents(); ++i)
  {
    els.push_back(D3DVERTEXELEMENT9());
    components_[i]->setVertexElement(els.back(), off);
    off += (BYTE)(components_[i]->size());
  }

  static D3DVERTEXELEMENT9 theEnd = D3DDECL_END();
  els[nComponents()] = theEnd;

  IDirect3DVertexDeclaration9* pDec = NULL;
  pDev->CreateVertexDeclaration(&(els[0]), &pDec);
  return pDec;
}
```

There are four `static_asserts` (see Boost Library Documentation in References). The first verifies that each component is a multiple of four bytes in size. The second and third verify that each level of recursive inheritance does not add any extra padding. They check that the size of each component class matches the sum of the sizes of its parts. The `static_asserts` are sanity checks, testing any assumptions about the behavior of the compiler. The compiler can add hidden members in many occasions. See *More Effective C++* (in References) for some examples.

The `EndList` class requires special handling. It is essential that it add nothing to the final composite vertex, whereas `sizeof(EndList)` must be nonzero (per section 1.8.5 in the ISO C++ Standard). Therefore, for cases involving `EndList`, the size of the `Composer` does not equal the sum of the parts. Fortunately, Boost's `type_traits` are perfect for detecting such a special case at compile time, as shown in the fourth `static_assert`.

The `packingCheck` method is called when the vertex description is created. The VB wrapper requires a description before a VB can be created. Hence, the packing check is called automatically as a side effect of creating a vertex buffer. The result is a robust system that does catch errors in practice.

The self-verification indicates that there is redundancy in the data. Does it imply that the system could be simplified? The problem is that the `D3DDECLTYPE` specifies more information than the compiler can statically deduce from a component class. The size of a component class can be taken with sizeof. But different types of components can have the same size. For example, `D3DDECLTYPE_FLOAT1`, `D3DDECLTYPE_D3DCOLOR`, and `D3DDECLTYPE_SHORT2N` all have a size of four bytes. It is possible to imagine Byzantine syntaxes that could deduce the `D3DDECLTYPE` from the component, but the extra complexity would probably complicate the framework rather than simplify it.

LISTING 3. The constructor for a vertex-buffer wrapper class.

```
class VertexBuffer
{
public:
    VertexBuffer(IDirect3DDevice9*,
        const VertexDescription&, size_t);
private:
    IDirect3DVertexBuffer9* pVB_;
};

VertexBuffer::VertexBuffer(IDirect3DDevice9* pDev,
    const VertexDescription& desc, size_t nVtx):
    pVB_(NULL)
{
    const size_t nBytes = nVtx * desc.sizeofVertex();
    pDev->CreateVertexBuffer(nBytes, 0, 0,
        D3DPOOL_MANAGED, &pVB_);
}
```

LISTING 4. Asserts verify self-consistency, helping to make the framework robust.

```
template <class C, class V>
void Composer<C,V>::packingCheck()
{
   assert(sizeof(C) ==
        sizeofVertexFieldType(C::description().type()));
   assert(sizeof(C) == C::description().size());

   BOOST_STATIC_ASSERT(sizeof(C) % 4 == 0);

   BOOST_STATIC_ASSERT((boost::is_same<V,EndList>::value ||
                  (sizeof(Composer<C,V>) ==
                  sizeof(V) + sizeof(C))));
   BOOST_STATIC_ASSERT((boost::is_same<V,EndList>::value ==
                  (sizeof(Composer<C,V>) == sizeof(C))));
   BOOST_STATIC_ASSERT(boost::is_empty<EndList>::value);
}
```

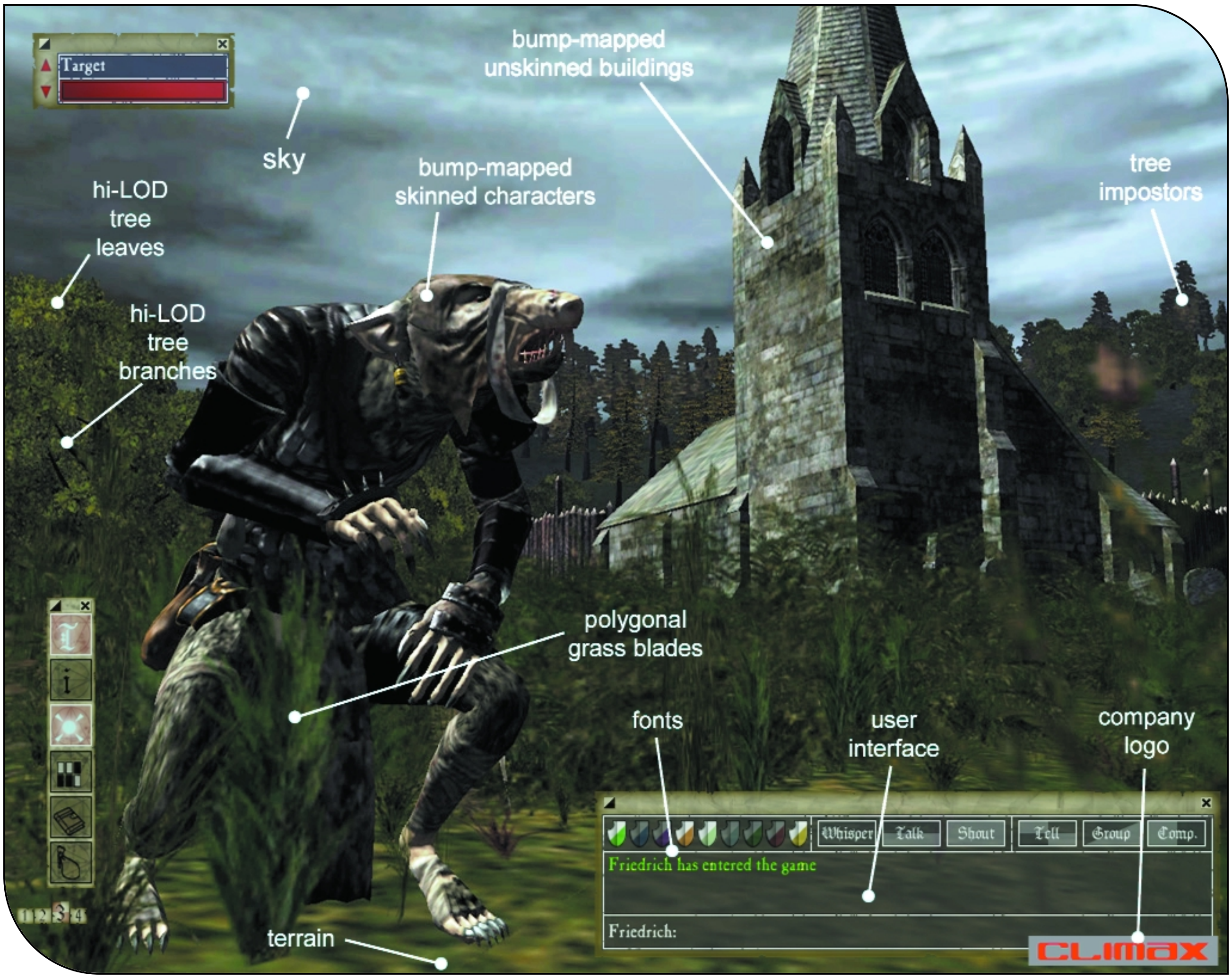


FIGURE 2. A screenshot from WARHAMMER ONLINE showing the different vertex types used in the engine.

Uses: Overloading Tricks

Each component in a composite vertex is a class. Implicit upcasts and overloading can be used to match vertex components against processing functions. An example is probably the best explanation.

In WARHAMMER ONLINE, vertex buffers are filled with data by a model importer. The imported models contain "fat" vertices. They include all the data that might be used by all possible vertices. Overloaded functions copy subsets of a fat vertex to the vertices used by the engine. Listing 5 shows that each component class matches a copying function. Where a destination vertex does not contain an optional component, it matches an empty function that takes a variable number of arguments. (The ellipsis is always a worse match per section 13.3.3.2 of the International Standard 14882 for C++.) Hence, a single template function can be used to create and fill all the supported vertex buffers.

Extending the Framework

The framework should provide a rich set of vertex components. Client programmers should not lack standard components. It is meaningful for the framework to provide components corresponding to all the standard fixed-function FVF components. These help make the whole system more robust and easy to use, because they are written and debugged once.

However, the framework must also be extensible. It cannot provide all possible vertex components, nor should it. Users can follow a simple recipe to implement their own, which can then be freely mixed and matched with the standard ones.

Examples

The WARHAMMER ONLINE engine uses many nonstandard vertex components. Many proprietary technologies and shaders add unique value to the product. Most are supported by their own vertex types. Figure 2 shows a screenshot of the game running and its various vertex types.

Of the objects shown, many require custom vertex components. The terrain uses one vertex type. A proprietary LODing and morphing algorithm requires custom components. The polygonal grass blades likewise use custom components for a proprietary LODing algorithm.

The trees use five vertex types. Custom components control the animation due to wind, billboarding, and the LOD algorithm. The trees are rendered using the Speedtree library (see

References), but the vertex data is still stored using this framework.

The skinned characters and unskinned buildings are both DOT3 bumpmapped. Their vertices are built entirely from standard components; likewise for the sky. Allowing for colored and uncolored types, these account for six types of vertex.

The fonts, user-interface widgets, and company logo use custom vertex components for 2D, unlit rendering.

In all, there are 22 different vertex types and the number is still growing. (A few are not visible in the example image, including a particle system, test harness data, and debug options.) The framework has been in use for about 18 months, and experience shows that it is trivially easy to add many vertex types to the engine.

Comparison with Alternative Approaches

The Microsoft Direct3D samples set the baseline for vertex buffer usage. A comparison is instructive. Two of Microsoft's samples have been rewritten to use the template code presented here and compared in Table 1. These are Fur and BumpSelfShadow.

Converting to use templates simplifies the client code. The reduction in the numbers of lines demonstrates that some tasks are automated and repetition is minimized.

The conversion process clearly demonstrated that the robust error checking features work in practice. I made a couple of typos while converting the samples. All were caught by the asserts.

The approach described here only works if the vertex types are known at compile time. If artists write shaders and can control vertex data formats then a more dynamic system may be necessary. (In our engine, all the details of vertex data are hidden from the artists by exporter software.) It would be possible to create a vertex buffer without requiring a compile-time class to represent the vertex. We prefer to leverage the C++ compiler. Compile-time methods have so far worked for every example in WARHAMMER ONLINE.

Sample	EXE Size (Kbytes)		Lines of Code	
	Before	After	Before	After
BumpSelf-Shadow	608	620	1864	1856
Fur	896	908	954	924

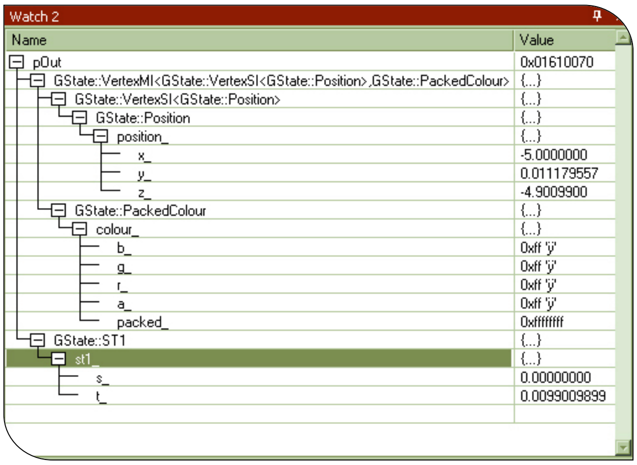
TABLE 1. The effects of applying the template classes to Microsoft samples.

LISTING 5. Function overloading provides type-safe vertex initialization.

```
void copyPosition(...) {}
void copyPosition(Position& dst,
        const FatImportedVertex& src)
{
        dst.position_ = src.position_;
}

void copyColor(...) {}
void copyColor(Color& dst, const FatImportedVertex& src)
{
        dst.color_ = src.color_;
}

// And likewise for TexCoords, Normal, and
// DOT3 basis vectors.

template <class V>
void copyVertex(V& dst, const FatImportedVertex& src)
{
        copyPosition(dstVtx, src);
        copyTexCoords(dstVtx, src);
        copyNormal(dstVtx, src);
        copyColor(dstVtx, src);
        copySBasis(dstVtx, src);
        copyTBasis(dstVtx, src);
}
```

Performance

There is an execution cost for building a `VertexDescription`. The templates generate a recursive set of function calls. (They don't get inlined by my compiler.) The time taken to execute these could be measured, however, the cost of creating a D3D vertex buffer is high. Relative to the D3D task, any time taken by the vertex framework code is presumed insignificant.



```
Watch 2
Name                                                                      Value
pOut                                                                      0x01610070
  GState::VertexMI<GState::VertexSI<GState::Position>,GState::PackedColour>  {...}
    GState::VertexSI<GState::Position>                                    {...}
      GState::Position                                                    {...}
        position_                                                         {...}
          x_                                                              -5.0000000
          y_                                                              0.011179557
          z_                                                              -4.9009900
    GState::PackedColour                                                  {...}
      colour_                                                             {...}
        b_                                                                0xff 'ÿ'
        g_                                                                0xff 'ÿ'
        r_                                                                0xff 'ÿ'
        a_                                                                0xff 'ÿ'
        packed_                                                           0xffffffff
  GState::ST1                                                             {...}
    st1_                                                                  {...}
      s_                                                                  0.00000000
      t_                                                                  0.0099009899
```

FIGURE 3. The multiple inheritance obfuscates the display of vertices in the debugger.

Code size is a more interesting factor. Templates are notorious for creating code bloat. However, the Composer template has only two methods, and each of those is small. Table 1 shows how the size of the executables changed when the Microsoft samples were converted to use template vertices. Each executable increases in size by a few Kbytes, but the cost is commensurate with the benefits.

In both cases, the frame rate was unchanged (quite predictably).

Wrap Up

This article has shown that Direct3D vertex buffers can be simplified using various C++ template techniques. However, there are drawbacks. The templates introduce a complex syntax where developers might normally expect a few C-style structs. Not all programmers have such familiarity with templates, so there may be a learning curve.

The same complex structures are also a problem in the debugger. Figure 3 shows a screenshot of a three-component vertex in Microsoft Visual Studio. Each level of inheritance is displayed as a nested class. In order to examine the members of a vertex, it is necessary to drill down through many displayed levels. However, there are other ways to debug the vertex data. It is always available in a more convenient format at the point at which the vertex buffer is initialized. It can be examined as the VB is filled. Alternatively, the data can be viewed in a shader debugger. A shader debugger is arguably preferable because debugging the shader and its input data go hand-in-hand. In practice, we find that debugging the vertex data is never a problem.

Some compilers may also have problems digesting the template syntax. Widespread support for default template parameters is relatively new. For older compilers, there are workarounds. We have successfully used the technique with three versions of Microsoft's Visual C++ (versions 6.0, 7.0, and 7.1).

I hope that this article shows that the benefits outweigh these disadvantages. The main benefit — automated robustness — is fourfold.

First, the vertex declaration is generated automatically merely by defining a vertex class. The D3D declaration is guaranteed to match the vertex class. Second, wherever possible, static type checking is used to increase robustness. The template VB wrapper hides `void*` data pointers completely. Third, vertex buffer creation is integrated with the vertex description class. A vertex buffer is guaranteed to be created with the correct element size. Finally, the fourth benefit is that compile-time and run-time asserts are used to verify that the compiler and client programmer both behave as expected.

Programmer learning curve notwithstanding, the framework makes it trivial to create a wide variety of new vertices. Only a few lines of code are required per vertex.

In a small way, it is close to the reuse ideal. Client programmers can reuse predefined components but also freely mix them with their own extensions. It raises the level of abstraction and enables programmers to concentrate on more interesting issues: writing ingenious shaders and gorgeous special effects. Eighteen months of use developing a cutting-edge game engine demonstrate that it works in practice.

SAMPLE CODE

The code listings have been expanded to the point where they form a working library. The library provides enough functionality to convert the Microsoft Direct3D samples referred to in Table 1. It is available for download from www.gdmag.com.

ACKNOWLEDGEMENTS

Thanks to JJ and Justin for valuable suggestions, and to JJ for actually implementing a couple of the key features when I just left "TBD" comments in the code. Thanks to Ed for feedback from a non-D3D perspective. The high-resolution creature images were provided by Toby and Neil.

REFERENCES

C++ TEMPLATE METAPROGRAMMING
Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison Wesley, 2001.
www.moderncppdesign.com

BOOST LIBRARY DOCUMENTATION
www.boost.org

D3D DOCUMENTATION
www.microsoft.com/windows/directx/default.aspx and
http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28000410

Wolfgang Engel, ed. Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks, Wordware, 2002.
www.shaderx.com

Meyers, Scott. More Effective C++. Addison Wesley, 1995. Item 24.

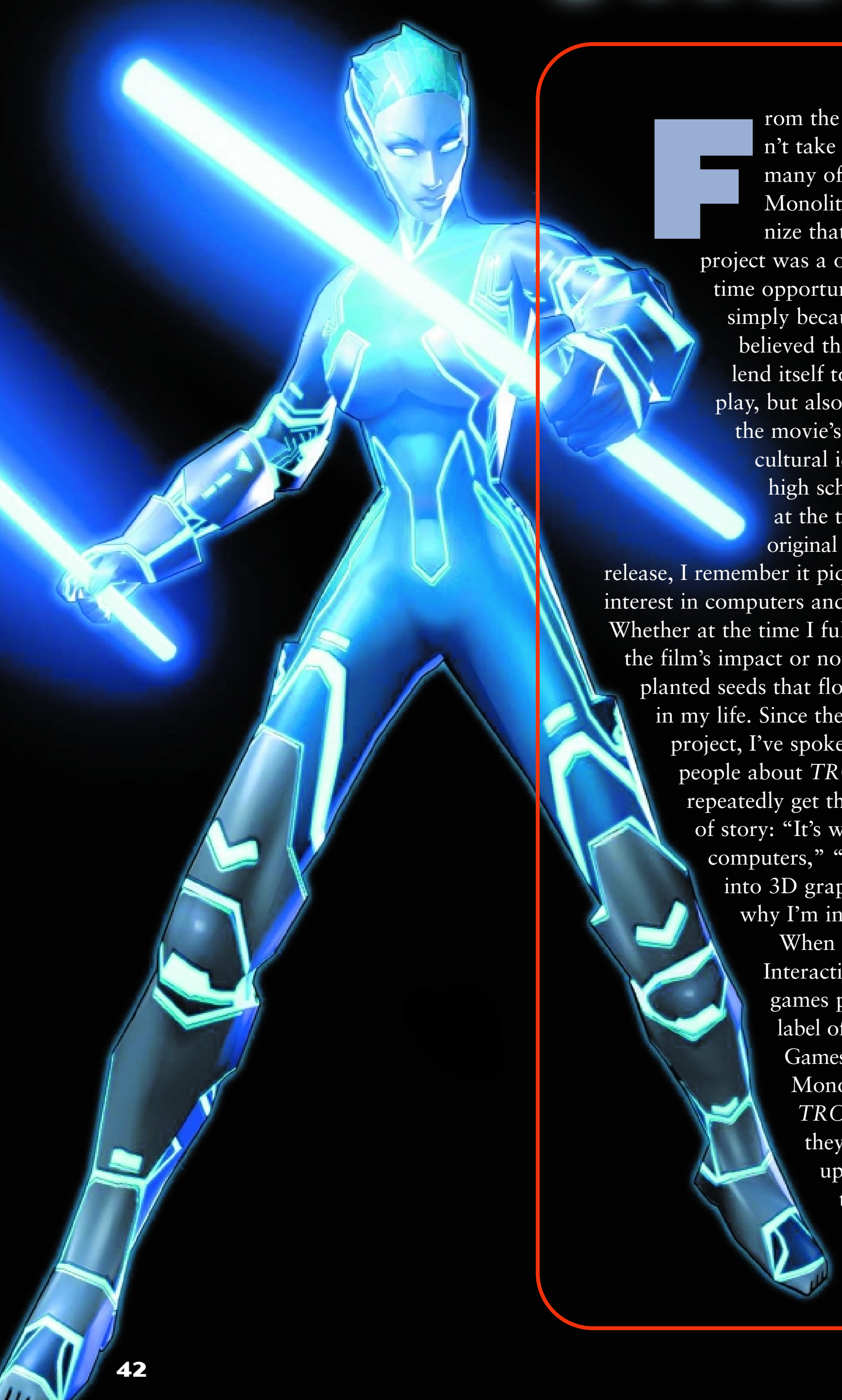
INTERNATIONAL STANDARD 14882 - PROGRAMMING LANGUAGE C++
www.ncits.org/cplusplus.htm

THE SPEEDTREE SDK
www.idvinc.com
also see this issue's Speedtree review, page 12

WARHAMMER ONLINE
www.warhammeronline.co.uk

frank rooke

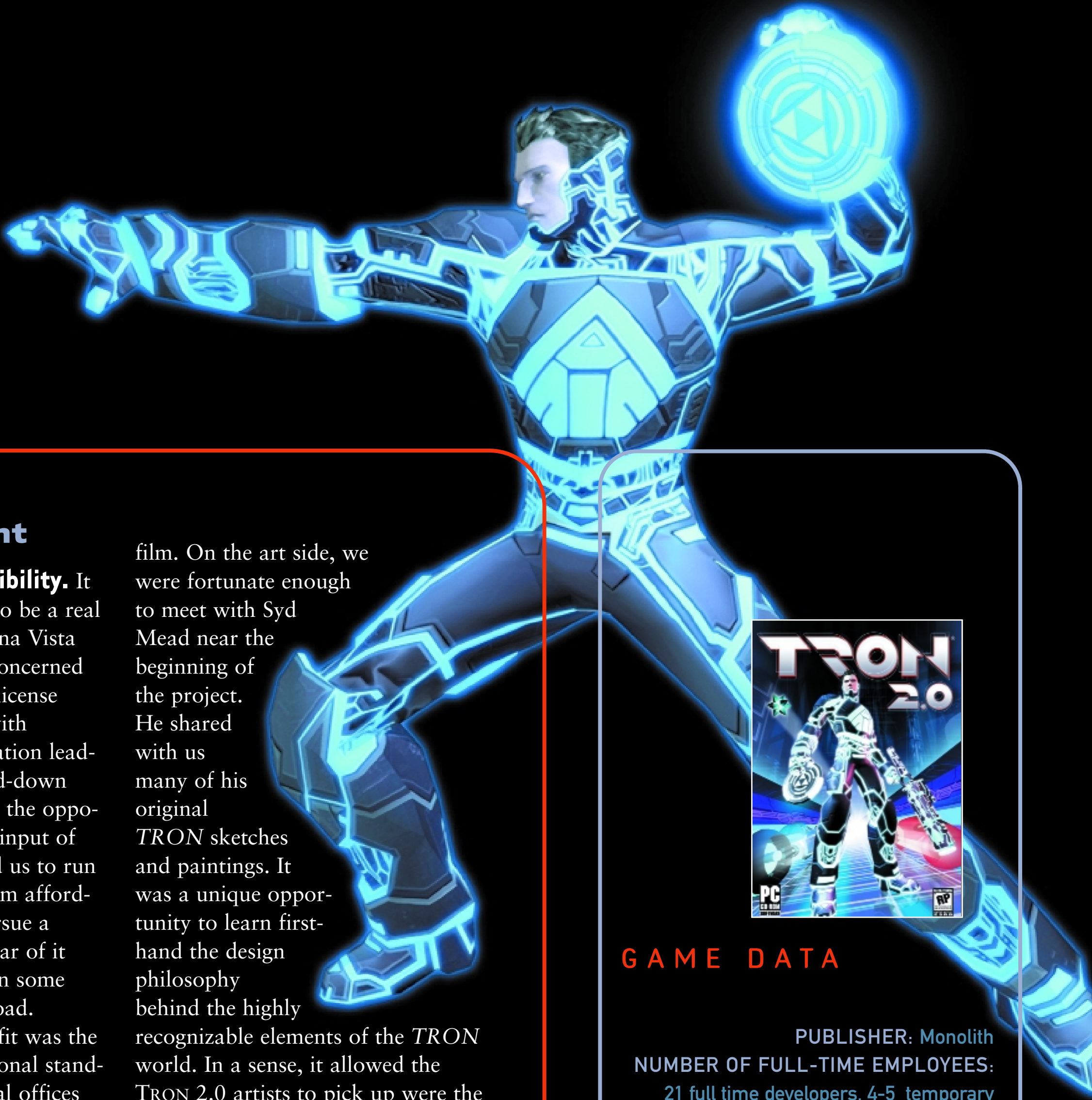
Monolith's TRON 2.0



From the start, it didn't take long for many of us at Monolith to recognize that a *TRON* project was a once-in-a-lifetime opportunity, not simply because we believed the film would lend itself to great gameplay, but also because of the movie's status as a cultural icon. As a high school student at the time of the original theatrical release, I remember it piquing my interest in computers and videogames. Whether at the time I fully realized the film's impact or not, it certainly planted seeds that flourished later in my life. Since the start of the project, I've spoken to many people about *TRON*, and I repeatedly get the same kind of story: "It's why I'm into computers," "It's why I'm into 3D graphics," "It's why I'm into gaming."

When Buena Vista Interactive, the core games publishing label of Buena Vista Games, approached Monolith with the *TRON* project, they were quite up-front about the challenges facing the franchise. While everyone readily agreed it would make a great computer game, generating interest for a title based on a 20-year-old cult classic film that was released ahead of its time might be difficult. Regardless, the project moved forward with great enthusiasm from both Monolith and Buena Vista Interactive. The fact that the game could possibly pave the way for a new *TRON* film and reignite the franchise was very exciting, injecting a unique motivation into the project that Monolith didn't take lightly.

Overall, TRON 2.0 is a first-person action game that takes place in the digital universe established by the 1982 film *TRON*. It's important to note that the game does not follow the events seen in the film. Instead, it is a spiritual sibling, or something of a sequel. The core premise of a society mirroring our own that exists in the computer remains intact, as does the phenomenon of a human transporting (or digitizing, as we say in the game) into the computer. Beyond that, the TRON 2.0 universe breaks new ground. Analogies, metaphors, and social consequences reflect how we understand and position computers in our lives today as well as where they may be in the near future. The game tells only one story of a hundred possible stories, making the *TRON* universe much like the *Star Wars* and *Star Trek* universes in that respect. It's this singular quality that makes the *TRON* franchise timeless.



What Went Right

1. Publisher compatibility. It was and continues to be a real pleasure working with Buena Vista Games. We were initially concerned that the constraints of the license would be overwhelming, with minute-level detail examination leading to a potentially watered-down game. However, it was just the opposite. While BVG had great input of their own, they encouraged us to run with our ideas. This freedom afforded us the confidence to pursue a game design without the fear of it changing or being altered in some obtuse fashion down the road.

Another peripheral benefit was the publisher's strong international standing. There are BVG regional offices across the world. Particularly noteworthy are those in Europe. From the onset of the project we had direct contact with the very people involved with press, retailers, and consumers across multiple European regions. From a design point of view, this exposure to non-U.S. markets was enlightening and useful. It's impossible to be all things to all people, but it is good design practice to consider the entire breadth of your target audience. TRON 2.0 is more accessible and dynamic because of it.

Lastly, BVG granted us access to the talent involved in the original film. On the art side, we were fortunate enough to meet with Syd Mead near the beginning of the project. He shared with us many of his original *TRON* sketches and paintings. It was a unique opportunity to learn firsthand the design philosophy behind the highly recognizable elements of the *TRON* world. In a sense, it allowed the TRON 2.0 artists to pick up were the film left off. Although the game achieves an overall look that is more detailed and colorful than the film, the consistency in the overall aesthetics between the two projects remains credible. Mead also contributed the new super light cycle, an exclusive design just for the game. Both Monolith and BVG agreed that it seemed appropriate, not to mention cool, to have the creator of the original light cycle design the next incarnation of the iconic bike.

Besides Syd Mead, the team had access to special effects director Richard Taylor and *TRON* creator

FRANK ROOKE | *Frank has been with Monolith four years and was the lead game designer on TRON 2.0. Prior to that he was a level designer for NO ONE LIVES FOREVER and lead level designer for the PS2 port. Before he went to work in games Frank was an interior and architectural designer in San Francisco.*

GAME DATA

PUBLISHER: Monolith
NUMBER OF FULL-TIME EMPLOYEES: 21 full time developers, 4-5 temporary developers pulled in as needed, plus the use of Monolith's internal sound, music and motion capture facilities and personnel.

NUMBER OF CONTRACTORS: Cinematic music scoring, motion capture actors, voice actors.
LENGTH OF DEVELOPMENT: 2 years
RELEASE DATE: August 26, 2003
TARGET PLATFORM: PC
DEVELOPMENT HARDWARE: Pentium 1.0-2.0GHz machines with 256-512 MB RAM Geforce 1-4 video cards

DEVELOPMENT SOFTWARE: Lithtech DEdit/ModelEdit, Microsoft Visual Studio (C++), Photoshop, Maya, Editplus 2
NOTABLE TECHNOLOGIES: Lithtech Jupiter Development System
PROJECT SIZE: 2,400 files, 853,300 lines of code

Steven Lisberger to review progress of the game. Taylor, on one occasion, popped into the Monolith office and provided some very helpful feedback regarding lighting and camera movement. On the acting side, Bruce Boxleitner and Cindy Morgan lent their voices to the game. Most notably, Boxleitner reprised his role as Alan Bradley.

2. Identifying iconic elements from the film. We asked ourselves, what were the core elements that provided *TRON* with its unique identity? Not surprisingly, we immediately isolated the disc and light cycle as iconic elements from the movie and marked them as mandatory features for the game. However, once we started looking past the obvious, we were a taken aback by the sheer quantity of other essential *TRON* components. To compound the issue, it became evident that different people — meaning various people on the team, at BVG, in the press, and at *TRON* fan sites — all isolated different elements or events from the movie as true *TRON* moments. What began as a simple checklist became a forum of discussion that never really concluded until the completion of the game.

To get a handle on the situation, we started prioritizing signature TRON components by how they supported gameplay and to what extent they propagated the *TRON* identity. We then discussed how to mature these concepts to meet the demands of a contemporary game. What we ended up with is a working mix of old and new — recognizable yet fresh. The combat component of the game still revolves around the disc, but it can now be upgraded. Environments retain the glowing, outlined look but with increased vibrancy and complexity. The story is new but resembles the original through the use of playful analogies, techie metaphors, and light-hearted humor — all hallmarks of the original script. And finally, memorable entities such as Bit, Tanks, and Recognizers make appearances but with altered functionality to represent the passage of time.

Concept art for the character Thorne.

Concept art and in-game shot of Jet Bradley in the real world.

We avoided simply translating the film directly into a game. It took significant effort to advance the *TRON* universe beyond the safety of the film. Setting out to improve iconic elements is always risky, but as a team we agreed not to take the easy road and short-change the property's potential by doing the bare minimum. Solely relying on the *TRON* name to sell units was not our strategy.

3. No movie license curse. It's a common belief that movie license–based games are substandard. How many times do we see game reviews with the comment "Game X is just another mediocre game based on a movie." We did not want TRON 2.0 to be another movie tie-in casualty. Not only would it be bad for Monolith's reputation, but we genuinely didn't want to waste the opportunity. TRON 2.0 needed to be able to stand on its own as a fun, engaging, and intelligent game, regardless of its lineage.

To help realize this goal, we began work TRON 2.0 as we do all our projects by reviewing successes and failures in our previous titles or similar titles so we could learn from past errors. TRON 2.0 is fully contextual to the *TRON* universe yet iterative relative to past Monolith efforts. With solid game design fundamentals learned from past projects, we were left free to explore unique game mechanics, storytelling devices, and technical enhancements that pushed TRON 2.0 into new territory.

4. Sharing code. The TRON 2.0 team found itself in a unique position. The No One Lives Forever 2 (NOLF 2) team was roughly eight months ahead of the TRON 2.0 schedule. They carried most of the burden of developing Jupiter, the next-generation game systems and tools needed to make NOLF 2 a cutting-edge game. TRON 2.0 was slated to closely follow NOLF 2 and directly use the Jupiter engine.

Although there were trade-offs, sharing code development with NOLF 2 primarily allowed us the freedom to focus a greater amount of our resources toward content, new features, and gameplay. TRON 2.0 certainly had its share of engineering hurdles, such as the glow effect, light cycle technology, and of course the engineering to support all of TRON 2.0's varied game objects. However, we didn't have to worry about creating a new renderer or AI systems. Also, uncertainties that are usually attached to new technologies were for the most part already resolved. We simply learned the parameters of the engine and adjusted the scope of our game to fit.

5. Evolved art direction. TRON 2.0 has received praise for its colorful architecture, glowing streams of energy, and creative level design. Without a doubt, the artists and level designers on the TRON 2.0 team successfully captured the essence of *TRON*. Not only do the characters and environments look like those found in the movie but in some cases surpass them. The art direction of TRON 2.0 really stands out as one of the primary attributes of the game, especially with the recent trend toward hyperrealistic military games. TRON 2.0 is a fresh alternative.

The method the art team used to achieve the look of TRON 2.0 was grassroots in nature. During preproduction, the artists re-created many of the actual sets from the film to get the feel for *TRON*. From there, they evolved the look to represent how computers changed over the last 20 years. It's interesting to note that keyboards, monitors, and circuit boards have changed little over the years. But *TRON* is not about the literal interpretation of computers; it's about the abstract world inside, the world of programs, data, and energy. It is here where more significant advancements have been made, and translating that into three-dimensional architecture was the greater challenge. Unlike building recognizable architecture, such as a warehouse or a subway, the artists and level designers had to develop the means to communicate through an abstract language to express what a firewall or PDA looked like to a program.

The film also had a distinctive glow about it. Initially, we attempted to build the glow directly into the art by using layers of additive textures. However, that proved to be time consuming and somewhat inconsistent. Plus, it was not a practical solution for characters. Collaboration between Monolith and Nvidia engineers produced a technique that generated a glow effect that was processed in real time by essentially applying a second render pass with a blurred effect. Once we saw this for the first time, it was clear TRON 2.0 was going to have a very special look. The glow effect immediately became an item of note when discussing the game with the press.

What Went Wrong

1. Short on initial resources. TRON 2.0 got off to a pretty good start. A lot of enthusiasm, ideas, and excitement flowed through the office. However, the majority of the team was still busy wrapping up other projects, and only a core team of about four or five actually began preproduction work on TRON 2.0. This is a pretty common scenario for most developers. The pacing of production usually has a small team jump-starting the next project by writing documentation, doing concept art, prototyping, and so on.

Unfortunately, we failed to recognize a few issues that caused ripples later on in the production. These are lessons that we definitely plan to implement into future projects.

The most obvious was the lack of ramp-up time — not just in skills needed to use the new tools, but also in the mentality and spirit of the project. Past Monolith projects have been founded in more easily understood worlds. TRON 2.0 was different. It actually took some time and effort just to wrap our brains around what was TRON 2.0's reality. When production officially started and the entire team was in place, there were quite a few months where output was minimal.

Initical concept art for the character Jet, abandoned for the more-hevaily armored blue look.

Also, the team's understanding of the game design was affected. Scads of design documents were written up in an effort to explain the concept of the game to the team. This worked to some degree, but was not ideal. Simply telling the team what to do not only failed to tap into the wealth of talent they had to offer, but also created a division in their perception of investment. Being involved during the inception of an idea, or actually being the one who spearheads the idea itself, fully invests a person into that idea. It's key to have the entire team feel personally attached to the project and not just see it as a series of tasks to be completed.

We did have team meetings throughout the preproduction period, and it's not as if the design was written in a vacuum, but communication could have been better. The ownership of the design should have been more team oriented from the beginning rather than something they had to grow to accept over time.

2. Levels unplayable until later in project. It wasn't until very late in the schedule that the team was able to experience a full, playable level. Fortunately, the gameplay came together nicely and in some cases exceeded our expectations, but it would have been a disastrous situation had it not. Why it took so long for the game to hit its stride may again be attributed to our underutilized preproduction period.

As I mentioned in the previous section, it took the team a considerable amount of time to "find" *TRON* — to collectively drive toward a unified vision. We had the necessary talent, and there was no shortage of ideas or inspiration, but the caliber of game we wanted to make was elusive at the beginning. In addition, TRON 2.0 pre-production did not include a fleshed-out prototype. Light cycle racing, disc combat, and the functionality of the subroutine system all loomed as unknowns for too long. To benefit fully from a good idea is to learn how to exploit it. Having key components of our game remain unknown generated trepidation when it came to linking time and resources. When the game finally did come together and we could see firsthand what worked well, we had very little time to build and expand confidently on those successful game mechanics.

3. Linking projects. Under "What Went Right," I mentioned that TRON 2.0 benefited greatly from the fact that we used the Jupiter system, which was developed and tested during the production of NOLF 2. This was true, but there were a few hurdles that also had negative impact on production.

Although the Jupiter systems were extremely flexible, and even more so now, at the time they were tailored for a more realistic game like NOLF 2. To alter or add functionality for the TRON 2.0 project meant lots of tweaking. In addition, the team preferred to limit changes to game code, rather than rewriting core engine components unless absolutely necessary.

4. Loose review process. During our internal postproject evaluation, one topic that consistently cropped up was the inadequacy of our internal review process. Multiple elements drive the production of any project: an individual's task schedule, the milestone schedule with the publisher, the E3 demo and press schedule, and finally the team's internal progress evaluation. In a lot of ways the team's own evaluations can be the most critical, and it is here that TRON 2.0 suffered some bumpy times.

In the beginning, we had in place a rather loose review system. It served the purpose of keeping the team on schedule to meet the necessary publisher milestones. What it failed to do was verify the project's overall quality and playability in a timely fashion. It was a strange sensation knowing that we were progressing according to the schedule but worrying deep down that we may not be hitting the mark. We eventually started to evaluate closely the work done by each team member at regular intervals, creating priority lists of subtasks that we felt were necessary to complete before the scheduled task could be crossed off. Toward the final third of the project, we had in place a very comprehensive review process that included weekly and sometimes daily reviews, a cross feedback system from all disciplines on the team, and an aggressive commitment not to falter, regardless of everyone's tight schedules.

Establishing a strong internal review system was so beneficial during the final stretch of TRON 2.0 that it is now one of our primary management goals for our next project.

5. Commercial 3D software woes. Level design reached a crossroads during the production of TRON 2.0 when we began using a commercial 3D package to construct our levels. The increased power and flexibility allowed us to model and texture the unique environments found in the game. However, abandoning our game editor during this phase of construction presented gameplay-related issues later in the project.

Concept and final art for the character Mercury.

In the past, constructing the world in our proprietary game editor allowed us to bounce back and forth easily between building and testing. When working strictly in a commercial package, the testing of work required the designer to jump through a few hoops to test work. While not overly complicated, it did require time, and therefore testing was done less frequently. Consequently, levels had problems regarding scale, flow, polygon counts, and delayed functionality — all things that could have been easily checked and verified in the game editor.

A great deal of time was lost reworking levels to make them playable. Fortunately, now that our designers are more in tune with the workflow between commercial tools and our game editor, this issue is mostly a thing of the past.

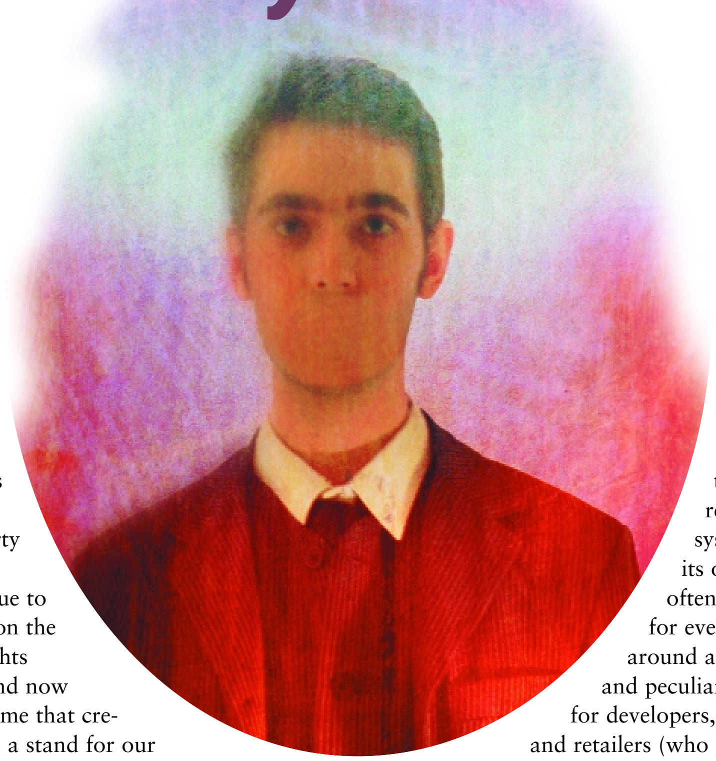
End of Line

One of the more remarkable things about TRON 2.0 is how an eclectic group of individuals came together and weathered many storms to prevail, not with just a finished product, but one of which we can be proud. The dedication and talent the team displayed excites uslooking ahead to our next project.

As far as the game itself in concerned, TRON 2.0 was truly the opportunity of a lifetime. I was personally unprepared for the amount of enthusiasm TRON 2.0 garnered from fans and from the press, both in North America and in Europe. We knew all along it was going to be a huge responsibility bringing the *TRON* franchise into the 21st century. Now that all the hard work is behind us, it's a thrill to know that what we've accomplished can potentially be the seeds from which future projects can grow. And as a huge fan of *TRON*, I can't wait to see the next tale told inside the world of computers.

Return

Regulation Is Everyone's Business



Thirty years into their popular existence and videogames continue to get beat up by the media, by politicians, and by activists purportedly out to save our culture. Thirty years is too long for videogames to continue to be constantly forced on the defensive. The old fights haven't gone away, and now more than ever it is time that creators collectively take a stand for our art form, our industry, and the careers we've built over a lifetime.

Simply put, there's crazy stuff going on out there. Thailand has recently implemented a 10 p.m. curfew on playing games in cafes. Germany's notorious "index" of blacklisted titles continues to grow. Greece tried to ban all digital game playing in one fell swoop. Afghanistan has shut down every last cybercafe as a means to preserve its cultural morals. And the state of Washington is set on protecting the health and safety of its law enforcement officials by regulating game sales to minors. Sadly, I could go on.

The perception that games are "bad" for us stubbornly persists, and we have yet to find effective ways to change people's minds on this issue. Game makers may be biased toward games' "good" qualities, but you'd be surprised how many developers simply don't care about the issue of public perception, don't have an informed opinion, or believe it is all a big waste of time — even to the extent of questioning the need to fight government regulations.

Sure, the headlines make us look bad: "Government working to protect; industry fighting for right to corrupt." Many of us are not comfortable confronting that image. The following are several common misperceptions developers hold about regulation and what our role, as creators, should be in fighting it.

Government regulation is no big deal, they're just reinforcing industry ratings. Wrong. None of the proposed bills are based on the ESRB ratings system. In fact, it's unconstitutional for the U.S. government to regulate or enforce a private ratings system. As such, each bill aims to set its own moral barometer and establish often vague metrics for what is acceptable for everyone to purchase and play. Dancing around a patchwork of content restrictions and peculiarities would be prohibitive not only for developers, but also for time-deprived parents and retailers (who are already working with an existing rating system).

Law X or Y doesn't seem so harmful, but fighting all of them makes us look bad. Each law and court case sets a precedent for the next. The St. Louis case where Judge Stephen Limbaugh ruled that games do not express ideas inspired both the Washington State bill and the reissue of Rep. Joe Baca's (D-Calif.) Protect Children from Video Game Sex and Violence Act in Congress. While Limbaugh's ruling was later overturned on appeal, reaffirming that games do express ideas and should be protected by the First Amendment, at least another dozen similar bills are in the works. For example, the state of South Carolina is looking to go one step further than Washington and regulate the sale of games depicting violence against law enforcement officials to all consumers, not just minors. That means you.

This doesn't affect me, it's the publisher's problem. Wrong again. While most publishers usually handle rating submissions and take the brunt of any backlash, the system has direct impact on developers. If retailers are unwilling to stock games with certain types of content (such as violence against law enforcement officials) for fear of running afoul of the law,

continued on page 55

continued from page 56

publishers will not have an outlet to sell such games and will therefore not fund game developers to make them. This is commonly referred to as the "chilling effect" of regulation. So, tough luck for any developer hoping to create the next crime-caper masterpiece.

I don't like or make violent games, so regulation is O.K. for me. Standing up for creative freedom isn't about fighting for the rights of any one specific game or developer. We need to stand up for the medium as a whole. Who are we (or anyone else for that matter) to judge what is good or bad? While I may not personally agree with some design choices, I strongly believe in developers' freedom of expression. Where will it end? The government's current fascination with violence may soon expand and put your nonviolent game square in their viewfinder.

Politicians are acting in their constituents' best interest, and there's nothing I can do. Historically, censorship and regulation have never truly been about the best interest of the people. If there's any doubt that some of these constituencies seem self-contradictory on the issue of violence, consider that Rep. Baca also voted to give gunmakers immunity from liability for crimes committed with their products — and that's just one example of many.

In light of these facts, it's clear that rolling over is the worst possible course of action for developers seeking to protect their craft and livelihoods.

Fortunately, there are several small and immediate steps we can all take. For starters, stay informed on the topic and read the news. You can head out to a local IGDA chapter meeting to discuss these issues with your peers. Try to attend at least one GDC session on these issues. The ESA also has a grassroots database (www.capitolconnect.com/esa) where you can sign up to follow what's going on in your town or state. Finally, write a friendly letter to your local and federal representatives explaining how you see your profession.

It is unconstitutional for the U.S. government to regulate or enforce a private ratings system.

In the bigger picture, resolve to push boundaries and innovate. Higher courts are reinforcing our view that games are an expressive medium worthy of the same free speech protections as other forms of art and entertainment. Let us not lose that respect nor give them excuses to question it. We need not put a stop to games with violence, but we need other avenues beyond violence as a design crutch.

Finally, have self-respect. As we all know, developing a game is a massively complex and creative endeavor. Take a stand when friends and family come down hard on games (yes, that means another attempt at explaining to your parents what exactly it is you do for a living). Better yet, don't back down when the media has got you cornered. Because the best defense is a good offense, the IGDA has prepared a simple set of talking points (www.igda.org/violence) every developer can use to talk him- or herself out of a corner.

Videogames belong at center stage.

JASON DELLA ROCCA | *Jason is the program director of the International Game Developers Association, working to build a unified development community and common industry voice on topics such as student outreach, concerns over game violence, diminishing the impact of exploitative patents, and increasing diversity. Contact him at jason@igda.org.*