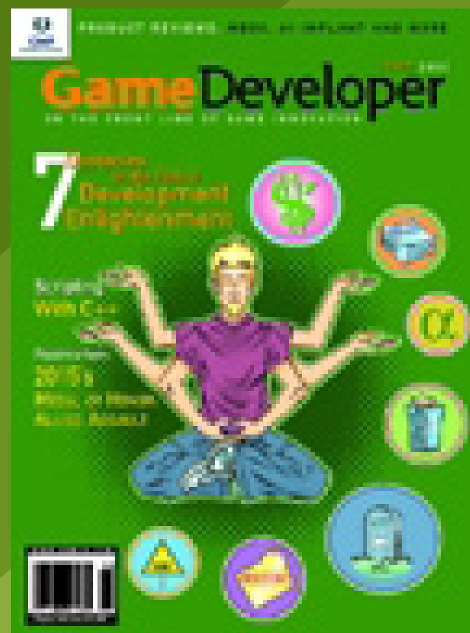
gd

GAME DEVELOPER MAGAZINE

JUNE 2002



UBM Tech

GAME PLAN

LETTER FROM THE EDITOR

Are We There Yet?

I noticed something wandering around the GDC Expo floor this past March. Middleware is making serious progress with game developers. At GDC '99, when Sony showed off the newly unveiled Playstation 2 and its adjunct Tools & Middleware program, "middleware" was the ubiquitous buzzword of the day. At the time it was a buzzword in the sense that there was little to show for all of the grandiose promises of painless development and seamless integration. Developers were characteristically skeptical.

In the three years since then, three new game consoles have entered the market, many PC developers have migrated to console development, and many middleware vendors have been slowly and painstakingly improving their tools. The smart ones have listened carefully to developer feedback and feature requests, and worked toward better integration with other vendors' offerings. Many of those who didn't have quietly departed this game development vale of tears.

Whereas up until recently technology licensing for AAA titles was far more the exception than the rule, developer trepidation toward middleware seems to be on the decline as more and more commercially successful titles are released that have been developed with various kinds of licensed technology. Three of our last five Postmortems have featured games that licensed crucial engine technology; February's DARK AGE OF CAMELOT and May's FREEDOM FORCE both used the NetImmerse toolkit; this month's MEDAL OF HONOR: ALLIED ASSAULT was built on QUAKE 3. Mike Milliger, lead programmer at 2015 and the author of this month's Postmortem, identifies in his article a trend in technology licensing with intriguing implications: that the more prized skill among game developers is now in evaluating, choosing, and integrating licensable technologies, not in developing new ones.

Whether you agree with such an assessment or not, it's yet another opportunity to look around and ask, where are we and what are we doing? Budgets can't go much higher if profit is to remain a goal (yes, we said that when they first broke a million dollars, and look where we are now), development times can't get too much longer (the risk of permanently losing our industry's top talent increases with every thankless, death-march crunch mode), and Christmas still comes but once a year.

The current generation of consoles has presented some interesting trends so far. There is already a whole lot of software out on the market showcasing these new hardware platforms, and all these games need to be able to differentiate themselves in a crowded field. But graphical gewgaw doesn't seem to be what's resonating with mass market consumers or capturing their imagination — it's gameplay. Whether it's a brand-new game experience or a sublime refinement of an existing theme, gameplay is the front on which this generation of consoles will wage war, many believe. PCs will be way out in the graphical lead soon enough anyway.

Here we are at a point where many respected, successful, and innovative games have come out using licensed technology, and taking the middleware plunge is getting easier. Essential elements such as refining control mechanisms (critical in crowded console genres), online and multiplayer components (for adding value), and systems integration (to streamline development) can benefit when programmers are liberated from the onus of building an engine from scratch. The result can be a more refined overall experience in roughly the same amount of development time. Most advantageous is that more and better tools simply give developers that many more options: build it if that's your core focus and talent, buy it if you know your game will benefit from efforts spent elsewhere. More options mean better odds at merging creative vision with commercial reality.

Jennifer Olsen
Editor-In-Chief

GameDeveloper

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

Publisher
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL

Editor-In-Chief
Jennifer Olsen *jolsen@cmp.com*

Managing Editor
Everard Strong *estrong@cmp.com*

Production Editor
Olga Zundel *ozundel@cmp.com*

Product Review Editor
Daniel Huebner *dan@gamasutra.com*

Art Director
Elizabeth von Büdingen *evonbudingen@cmp.com*

Editor-At-Large
Chris Hecker *checker@d6.com*

Contributing Editors
Jonathan Blow *jon@bolt-action.com*
Hayden Duvall *hayden@confounding-factor.com*
Noah Falstein *noah@theinspiracy.com*

Advisory Board
Hal Barwood LucasArts
Ellen Guon Beeman Beemania
Andy Gavin Naughty Dog
Joby Otero Luxoflux
Dave Pottinger Ensemble Studios
George Sanger Big Fat Inc.
Harvey Smith Ion Storm
Paul Steed WildTangent

ADVERTISING SALES

Director of Sales & Marketing
Greg Kerwin e: *gkerwin@cmp.com* t: 415.947.6218

National Sales Manager
Jennifer Orvik e: *jorvik@cmp.com* t: 415.947.6217

Senior Account Manager, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 415.947.6224

Account Manager, Northern California & Southeast
Susan Kirby e: *skirby@cmp.com* t: 415.947.6226

Account Manager, Recruitment
Raelene Maiben e: *rmaiben@cmp.com* t: 415.947.6225

Account Manager, Western Region, Silicon Valley & Asia
Craig Perreault e: *cperreault@cmp.com* t: 415.947.6223

Account Representative
Aaron Murawski e: *amurawski@cmp.com* t: 415.947.6227

ADVERTISING PRODUCTION

Vice President, Manufacturing Bill Amstutz
Advertising Production Coordinator Kevin Chanel
Reprints Cindy Zauss t: 909.698.1780

GAMA NETWORK MARKETING

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

CIRCULATION

Game Developer is BPA approved

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

SUBSCRIPTION SERVICES

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

INTERNATIONAL LICENSING INFORMATION

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

CMP MEDIA MANAGEMENT

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

GamaNetwork

SAYS YOU

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

Slerping Your Way to the Top

I enjoyed Jonathan Blow's April "Inner Product" column, "Inverse Kinematics with Joint Limits." I especially liked Listing 1, the "fastest simple-rotation-finder in the West." It had never occurred to me to use slerp to compute the square root of a quaternion. My question is, why bother to multiply all the components of the quaternion by 0.5*f* when it is normalized in the next step? In other words, we can improve the function further by omitting these multiplications.

Another optimization involves the call to `fast_normalize()`. Since the input vectors are unit length, it's possible to get a simpler formula for the squared length of the quaternion that we have to normalize. It is `(2 + 2 * dot)`, where `dot` is the inner product of the vectors. So it's overkill to call `q->squared_length()`, as happens in `fast_normalize`. Computing the dot product is usually pretty slow, so this could be worthwhile. The downside is you'd have to create a special version of `fast_normalize()` to get this optimization.

I really learned from this column. It hadn't occurred to me to use slerp to get the square root of a quaternion. Jon has some really good ideas, and I look forward to his future columns.

Bill Budge
The 3DO Company
via e-mail

Jonathan Blow responds: *Yep, indeed that works and, in fact we can wrap all this up into one optimization. So we want to find*

$$0.5^2(q_x^2 + q_y^2 + q_z^2 + (1 + q_w)^2)$$

but since the components of the quaternion are the results of dot and cross products, we can rewrite this as

$$0.25(\sin^2\theta + (1 + \cos\theta)^2).$$

A trigonometric identity gives us

$$0.25(1 - \cos^2\theta + (1 + \cos\theta)^2)$$

and we can simplify this down to $0.25(2 + 2d)$*, where* $d = \cos\theta$ *is the dot product of the two vectors (what Bill is talking about). Multiplying the 0.25 into that gives us* $0.5 + 0.5d$ *as the squared length of the quaternion, which we pass into the inverse square root approximator.*

But that approximator already has additive and multiplicative terms, so we can factor these 0.5 terms into there, and make a new function that is just a factor of d*. And there we go. I think it's justifiable to have a special version of* `fast_normalize`*, assuming that a fast version of* `simple_rotation` *is important to your app.*

By the time we've done all this, we've saved something like eight multiplications and four additions over the original function. For people who are used to calling functions like acos to do quaternion stuff, that doesn't sound like very much; but since we'd already gotten this task down to a couple handfuls of adds and multiplies, Bill's improvements eliminate a large percentage of the function's work, and thus constitute a major optimization.

Thanks to Bill for the optimizations and the nice comments about the column.

Getting Emotional About Games

Thanks to David Freeman for his insightful article, "Four Ways to Use Symbols to Add Emotional Depth to Games" (February 2002).

I must admit, glancing through the mag, I was set to dismiss the article as yet another "helpful technique" to legitimize what seems to be very expensive and shallow thumb-twiddling.

Don't get me wrong; I work in the game industry, and I am grateful to have this creative outlet, to say nothing of it being my rice bowl. If people take David's message to heart, I'll never have to retire! For what they're worth here are some of my observations:

First, I agree with all of David's rationale for increasing the emotional depth of games. For years, the industry has touted improved hand-eye coordination as videogames' single point of redeeming social value, and it has therefore been beset with criticisms of moral and intellectual bankruptcy. No wonder. I'm glad to hear a voice for subtlety and spirituality amidst the cacophony of screaming hylics.

Second, it seems that the notion of adding emotional depth to enrich the game experience might also have a beneficial psychological effect on the player. For instance, in the case of the Subplot symbology to identify a character's FLBWs and ensuing growth, if players' FLBWs identify with the character, they may be tempted to enact their own catharses in real life. It's a stretch, I know, but we all identify with characters in movies all the time, and role-playing is a bonafide psychological tool for operant conditioning, so deep game-playing could enable players to realize more about themselves. This point is especially interesting to me. I often wonder why we all are so incapable of seeing our own FLBWs. And wouldn't it be helpful to be able to practice growing and evolving instead of just practicing our aim?

Third, I was intrigued by David's comments about the effectiveness of symbology being linked to its unobtrusiveness. Individual writers and artists create imagery, each from their own composite vision. Sometimes they will insert a symbol to enhance, but more often the symbology is resident in the original vision. It may even exist without the artist's awareness. It's different with a team making a videogame. David's points about inserting symbology are well taken. Filmmaking is a committee effort also, but the studios have been doing it longer than game-makers.

Steve High
Sony Computer Entertainment America
via e-mail

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

INDUSTRY WATCH

THE BUZZ ABOUT THE GAME BIZ | *daniel huebner*

Activision has acquired Shaba, developers of Shaun Murray's Pro Wakeboarder.

Awash in red Ink, Interplay settles its feuds. The turmoil at Interplay continues unabated, as the company reported sharp declines in fourth-quarter and full-year revenues for 2001. Interplay brought in fourth-quarter net revenues of $21.5 million, a full 30 percent drop from the previous year, leading to a net loss of $4.9 million for the quarter. The company posted a loss of $4.8 million in the same period one year ago. For the full year, Interplay's revenues fell to $57.8 million, a 45 percent drop from the previous year, pushing the company's net loss for 2001 to $46.3 million. Losses for the previous year totaled $12.1 million. Interplay attributes its poor overall performance to a weak release schedule — the company shipped only three games in the fourth quarter and managed just 10 in the entire year, combined with high product returns, and product mix overly dependent on low-margin PC games. Titus Interactive has increased its stake in Interplay, now owning 72.4 percent of capital stock.

The company did manage to follow up its bleak financials with some encouraging news by settling an outstanding dispute with former chairman and CEO Brian Fargo. Shortly after Fargo's resignation from both positions in late January, an Interplay company filing revealed that the struggling publisher was considering filing a suit against him for allegedly soliciting Interplay employees. Neither side revealed the nature of the settlement.

Creative acquires 3Dlabs, Activision grabs Shaba, Outrage goes to THQ. Creative Technology is acquiring 3Dlabs in a stock and cash transaction. Under the terms of the agreement, Creative will pick up 3Dlabs stock at $3.60 per share in a deal worth $170 million. Two-thirds of the shares will be converted to Creative stock while the balance will be converted to cash. 3Dlabs will continue to supply, support, and develop its professional graphics product lines, including Wildcat and Oxygen, and will carry on its standardization activities with OpenGL 2.0, OpenML, and embedded OpenGL in the Khronos Group and the Web3D Consortium. The transaction is subject to the approval of 3Dlabs shareholders.

Activision continues to buy up its more successful development partners, this time wrapping up a deal to acquire Shaun Murray's Pro Wakeboarder developers Shaba Games in an all-stock deal worth close to $7.4 million. Shaba and Activision previously worked on Mat Hoffman's Pro BMX and Tony Hawk's Pro Skater 3 for the PSX. Under the terms of the agreement, Shaba will become a wholly-owned subsidiary of Activision. Shaba's equity holders received 258,621 shares of common stock, valuing the deal at approximately $7.4 million. As part of the agreement, Shaba's key personnel will stay on under long-term contracts.

THQ added Michigan's Outrage Entertainment to its internal studio roster. The deal reunites Outrage with Volition under the THQ roof; both companies were created by the break-up of Descent developer Parallax Entertainment.

Vivendi takes on Battle.net clone. Blizzard Entertainment's parent company, Vivendi Universal, has filed a lawsuit against a company accused of operating an unauthorized Battle.net site. Vivendi filed suit in Eastern Missouri Federal Court in St. Louis against Internet Gateway, operator of The BNETD Project. The suit also names Tim Jung, one of BNETD's developers. While Internet Gateway maintains that BNETD is simply a Battle.net emulator designed by volunteers because of Battle.net's lack of reliability, Vivendi contends that BNETD violates several Vivendi copyrights and trademarks. The company had originally threatened action under the anti-circumvention provision of the Digital Millennium Copyright Act because BNETD doesn't perform a key check to ensure players are using legally purchased Blizzard games.

GTA 3 leads Take-Two turnaround. After taking a beating in the media and the markets due to financial irregularities and misreporting, Take-Two reported a stunning jump in first-quarter results. Take-Two reported first-quarter sales totaling $283 million and a net profit of $34.8 million; the company posted a profit of $8.2 million on sales of $158 million in the same period last year. Despite strong showing from other titles, including Max Payne and State of Emergency, Take-Two realized fully 41 percent of its sales from Grand Theft Auto 3.

Grand Theft Auto III helped pushed Take-Two's sales up in 2002.

UPCOMING EVENTS
CALENDAR

SIGGRAPH 2002
Henry B. Gonzales Convention Center
San Antonio, Tex.
July 21–26, 2002
Cost: $50–$950 (member and student discounts available)
www.siggraph.org/s2002

Digidesign's Mbox

by gene porfido

Over the past four or five years, those of us who have chosen our computers as a means to record music and sound effects have been blessed with an incredible choice of affordable software and hardware. Who remembers the original Sound Tools days? The first time I saw a sales rep manipulate a Yes song left me in awe. The cost was astronomical, even by that time's standards; my studio's first 600MB hard drive, which we picked up for Sound Tools, cost close to $4,000. Today, that would buy enough gigabytes to store everything I've recorded in the past 10 years.

With the advent of USB and Firewire, new avenues have opened up for connecting hardware, transportable storage (hot swappable at that). We've been granted an exceptional means to input and output signal to our computers. In the past we had sound cards or, with Macintoshes, the built-in 16-bit I/O. But thanks to companies like Digidesign, we no longer have to worry about quality I/O and software at an acceptable price.

It's All in the Box

Enter Mbox, a collaboration between two partner companies, Digidesign and famed British audio designer Focusrite, a name revered in pro audio circles. Digidesign is best known for its Pro Tools systems of hard-disk recording, but its basic professional set-up is expensive and overkill for simple home studio or multimedia needs. A few years back they released the Digi 001 with Pro Tools LE, an affordable I/O and software package that addressed the needs of most lower-budget users, but with up to 18 inputs/outputs, it's still a bit too much for some.

With Mbox, the two companies offer us a smaller package that exemplifies ease of use in a powerful system. Mbox has a small footprint, as it sits upright on your desk or computer table. Compact yet sturdy and well made, it's a nice design that takes up little space, looks great, and is eminently portable.

I/O, I/O, So Off to Record We Go

Mbox's front panel is simple and well laid out, showing the unit's underlying flexibility. There are four main knobs situated vertically on the front panel, similar in style to the ones in Focusrite's Green and Platinum series. The top two knobs control gain for the two input sources, the third is an input/output monitor mix (more on this later), and the last knob is the headphones level. The two input source gain knobs have a small button to the left that cycles through that channel's input options, and three LEDs to the right that correspond to whatever mode the input channel is in, with choices being mic (green), line (green), and instrument (yellow). Directly below these LEDs and adjacent to the gain knob is a red peak LED for each input source.

Beneath the two input channels, on each side of the panel, are two additional LED indicators. The left green LED lights up when an S/PDIF signal is present, acknowledging that Mbox's inputs are set to receive digital in as opposed to analog. The one on the right signifies proper USB connection, giving visual feedback that the Mac recognizes the Mbox and has loaded the drivers for it.

There have been a few software issues with Mac OS 9.1 and Mbox not properly loading its USB drivers (as I found out shortly after my first reboot), but a quick fix — unplugging the USB cable from the back of the box and then reconnecting it — works more often than not. When I kept getting error messages or peak lights on Mbox, I had to drag and drop the whole collection of Digiextensions a couple of times too. Digidesign is aware of the problem and is working toward a permanent fix. For the time being, they recommend upgrading to Mac OS 9.2 and downloading the latest drivers from their web site.

GENE PORFIDO | *Gene operates Smilin' Pig Productions, his own sound design and music company located in San Francisco.*

The third knob down, actually a mix/ratio knob, works by mixing your input source (the knob all the way to the left) with the playback from software (the knob all the way to the right). You can dial in any blend of the two, which gives a zero-latency response to your input signal; you're listening to the track through Mbox, not through software, similar to listening through a mixer channel instead of "off tape," so there's no noticeable delay between what's playing and what you hear. If Mbox handles all of your monitoring and input/output chores instead of routing through a console, this is a nice addition, perfect for the small home studio on a budget.

The fourth and last knob, controlling headphone levels, has a mono-sum button to its side so you can check for phase problems or just listen to a mono mix. There's a 1/8-inch mini-plug on the front for connecting applicable headphones.

The M Connection

The back of Mbox is where it all gets patched in. Two Focusrite mic pre's, highlights of the input channels, guarantee a good sound. They are multi-connector-type jacks and accept either XLR or 1/4-inch in the same socket. This allows Mbox to use only one connector on each of the two inputs, and gives the user the option of a line/instrument or mic.

Four 1/4-inch connectors are located above the input jacks, the bottom two being channel inserts that access the channel after the preamp and before the A to D converter. These Tip-Ring-Sleeves give the user the ability to patch in their favorite analog device, such as a compressor or an EQ. The remaining two 1/4-inch jacks are line outputs, typically a left and right out from your computer. These can be patched to a mixer, a home stereo, a power amp for monitoring, or sent to a two-track for mix-down.

Above the outputs are RCA-type S/PDIF connectors, with an in and an out. A 1/4-inch headphone jack is also offered, but keep in mind plugging into

this jack will negate audio to the front panel mini-plug headphone. The remaining items on the back include a single USB connector, and the phantom power switch providing 48 volts to both input channels simultaneously. Because it can potentially damage certain mics, most notably ribbon-type mics, it's a good idea to turn your volume down and unplug unused inputs when switching phantom power on and off.

That's Not All, Folks

Pro Tools 5.2 (version LE), included with Mbox, is the most popular hard-disk recording software in the world. But, since this isn't a Pro Tools review, I'll just add that this is a limited version of the fabled program, maxing out at 24 audio tracks. Still, Pro Tools LE is an exceptional recording package that will fulfill many people's needs, with plenty of included plug-ins for EQ, compression, and delays. Other plugs are available at an extra cost.

During the time I used Mbox, it met all my expectations and more. Perhaps its most notable feature is the fine tone of the input channels, a combination of the Focusrite mic pre's and Digidesign's 24-bit converters. There's plenty of routing; I used Mbox primarily with a small console, returning the two outputs in channels in the console for processing with live MIDI tracks. By plugging a number of powered and nonpowered mics and instruments into Mbox, I tested every input choice, and everything worked as I'd hoped.

Mbox can also be used with other sequencers or audio programs. Once the USB drivers are loaded, you can select Mbox in the Digidesign control panel, then select the Digidesign I/O from inside the software. I had absolutely no problems using Mbox with both Mark of the Unicorn's Digital Performer 3.0 and the included Pro Tools LE.

For a small studio that needs only one or two simultaneous inputs, Mbox is a charm. For a larger professional studio, Mbox offers two channels of quality audio in and out of your Mac. With its excellent sound quality, well-made hardware, and world-class software, the Mbox is hard to beat for its price (around $495 MSRP), and it's a great choice for someone seeking ease of portability. If you're looking to get into 24-bit recording, the Mbox is an exceptional way to get there.

MBOX ★★★★

STATS

DIGIDESIGN INC.
Daly City, Calif.
(650) 731-6300
www.digidesign.com

PRICE
$495 (MSRP)

SYSTEM REQUIREMENTS
Macintosh with factory-shipped USB port running Mac OS 9.1 or higher (OS X and Classic mode unsupported) with 128MB RAM (192MB recommended for newer Macs, virtual memory not supported), and CD-ROM drive for installation. Also Opcode OMS 2.3.8 and QuickTime 5, both supplied on Pro Tools CD.

PROS

1. Low price makes it very accessible.
2. Focusrite mic pre's.
3. Pro Tools software.

CONS

1. Some software hitches with OS 9.1.
2. No half rack or drive bay mounting scheme.
3. Pro Tools LE is not as powerful as regular Pro Tools.

BIOGRAPHIC'S AI IMPLANT PLUG-IN

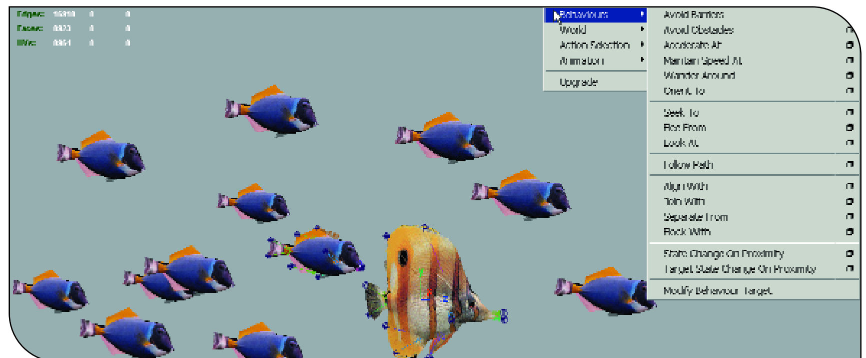
by kian bee ng

Biographic's AI Implant (currently available for Maya 3 and 4, with Lightwave and 3DS Max versions and an SDK planned) allows users to animate large scenes, by splitting them into manageable parts as necessary, while retaining control over the individual or grouping characters as an integrated whole.

AI Implant differentiates between two types of characters, autonomous and non-autonomous. An autonomous character exhibits intelligence in its course of action. The intelligence is governed by a set of behaviors that are assigned to it, and there are many behaviors you can assign. You can make the character flee, seek, wander around, avoid obstacles, follow a path, move with its group, separate from the group, and so on. In addition, each behavior is fully modifiable, allowing users to create detailed and specific animations.

Unlike autonomous characters, the AI solver does not influence non-autonomous characters. These are largely scene props, such as chairs, doors, or flags. Their presence serves to complete an interactive environment. Furthermore, you may also create objects that constrain the movement of the characters. These constrains include barriers, paths, and terrains. With barriers, characters may be set to avoid them; with paths, characters may be instructed to follow them; and with terrains, characters may be made to always stay on the surface geometry.

There are two types of solvers available within AI Implant, 2D and 3D. The 2D solver is for characters that are constrained on a 2D surface, and therefore more computationally efficient. The 3D solver is used when the characters move freely within the 3D world. Unlike other solvers in Maya that are default and fixed, AI Implant lets you create multiple solvers. This allows you to have different solvers for different character sets. You might bind a group of bugs on the ground with a 2D solver and assign a 3D solver to the flock of birds in the sky. Still, by combining the AI solvers with Maya's dynamic solvers, users can easily create a fascinating environment in which all objects exhibit movements that convincingly mimic the real world.



A group of fish moving in a 3D world and the list of behaviors in Biographic's AI Implant available to control them.

The heart of AI Implant lies in its binary decision tree. A binary decision tree acts like a simple switch: given a condition, go to state A or go to state B. With the binary decision tree, not only can you modify characters' behaviors, you can also use it to drive animation cycles or update a character's data.

AI Implant marks an important step toward managing large-scale crowd scenes, offering artists a viable means to create and control thousands of characters with ease. The only reservation I have is the hefty price tag; $5,000 might be easier to justify as later versions add more robust behavior options and run them faster. As it stands, small and independent studios with only occasional requirements to deliver large crowd animations may find it hard to justify the expenditure. Nonetheless, once more features and advanced behavioral controls are added, AI Implant may well become an indispensable tool.

Note: Pay-per-use licenses for AI Implant are available.

★★★½ | AI Implant 1 for Maya
Biographic Technlogies
www.ai-implant.com

Kian Bee Ng is a programmer on the Gran Turismo *series, writing tools and plug-ins for Maya.*

BERKELEY DB
BY SLEEPYCAT SOFTWARE INC.

C++ NETWORK PROGRAMMING, VOL. 1: MASTERING COMPLEXITY WITH ACE AND PATTERNS
BY DOUGLAS SCHMIDT & STEPHEN HUSTON

reviewed by crosbie fitch

Now that admitting to using middleware in your games is getting more respectable, the next step is admitting to using open source middleware. Don't worry, it's just like saying you use DirectX — it's free middleware and someone else developed it. While there are debates as to whether open source is really cheaper, at least you get to peruse and tweak the source code.

One of the ways that open source middleware vendors can bring in extra income is by publishing reference books or manuals concerning their middleware — even if this information is also freely available in online documentation (it seems even leading-edge coders still like the old fashioned ink-on-wood-pulp distribution medium).

I've picked two books, each corresponding to game components increasingly likely to be outsourced as games get ever larger in scale: the scenery database and the networking layer. You may find it hard to imagine using off-the-shelf components that didn't necessarily have spatially indexed, visibility-analyzed 3D

objects in mind when they were designed, but sometimes it might be expedient to knock something together that does the job before you get around to doing it properly (which may not be until after the game has shipped).

Berkeley DB addresses the database engine of the same name, while *C++ Network Programming* discusses the ACE (Adaptive Communications Environment) networking layer.

Each of these components can be used in your game statically, via DLL, or (because you have the source) in any other manner you see fit. As you might expect, and given that you need a typical OS and C compiler, most Windows and Unix type platforms are supported. However, whether or not it's easy to port to them, game consoles aren't really addressed.

Can you use them in proprietary game code? In the case of ACE you're in luck. It's subject to a BSD-style license, meaning it can be freely used in proprietary code even without disclosure. Berkeley DB is almost as available; rather, it has older versions that can be freely used in proprietary code (BSD license). However, in its recent, more powerful versions (say, multi-user), you're required to purchase licenses for proprietary use (though it's still free for use in open source code).

So while Berkeley DB and ACE might be worth looking into for your next game, are the books worth buying?

Berkeley DB is a pleasure to read. You come away feeling you've been told all you need to know about it, that the system has been lovingly crafted for demanding professionals such as yourself, and you get in the mood to download the code and build it into your game (or a few test harnesses first). The book gives you a very good overview of how Berkeley DB works, how it should be applied, how it fits into the platform (with considerations for the ambient operating environment), coding examples, building for different platforms (with FAQs), and even a test suite.

After all this, at page 245 (out of 642) you get API documentation for C, C++, Java and TCL. Database theory is briefly introduced, but Berkeley DB is really a core, scalable building block (hash tables, B-trees, simple numbered records, and persistent queues) from which you can build more complex database architectures, so you don't need to know about say, relational databases, unless you want to build one. And no, you don't need to learn SQL or anything like it either. This really is a neat, self-contained component. Buy the book and keep it by the bedside; it's a much better read than the documentation available online.

C++ Network Programming is a different reference in many ways. Patently trying to hijack the less cautious developer looking for general coverage of the issues and techniques, it's only when you read the subtitle, *Volume 1: Mastering Complexity with ACE and Patterns*, that you might suspect this book has a different agenda. With very little C++ coverage, and very little about network programming in general, it should really be titled *ACE — Adaptive Communication Environment*, and subtitled *A Cross-Platform Networking Layer, Abstracting Typical Facilities Required by Distributed Applications*.

If you've studied up on network protocol (see Comer and Stevens' *Internetworking with TCP/IP* series from Prentice Hall), the Sockets API (even WinSock), multi-threading, CORBA and/or COM, and Unix and Windows operating systems, then you're ready for cross-platform networking middleware such as ACE. Addressing the diversity of networking environments in-house is a bleak prospect: you're choosing from DirectPlay, a proprietary third-party networking middleware vendor, or ACE. If you're not ready for this book, don't read it. Unlike newbie-friendly Berkeley DB, ACE will burn your brain.

Because it's written in a formal academic/hardcore systems programming style, you need to know the underlying systems and historical precedents to have much of an idea of what ACE is trying to do. The style follows, but it overdoes the mantra of "(1) Tell them what you're going to tell them, (2) Tell them, (3) Tell them what you just told them." That is to say, a lot of each chapter spends time recapping what the previous chapter covered, introducing what the chapter is about to cover, covering it, summarizing what the chapter covered, and introducing the next chapter.

The book pats itself on the back a bit about its pioneering use of design patterns (façade, factory, and the like — all rather underwhelming). However, since it has "C++" in the title, I would've liked to have seen some evidence that the "object" design pattern was used somewhere (as in object oriented). Nevertheless, I tip my hat to ACE's laudable achievement in representing a cross-platform networking library. That's what's exciting here.

As a 268-page introduction to ACE (with no API reference section), the book is passable. It's a bit like an annotated reference manual and tutorial, except that the annotations are mixed in with the main text and it's not obvious when you're reading to which slant the information is directed — to someone interested in the design of ACE, to a designer or implementer of something like ACE, to a user of ACE, to someone interested in general networking issues, and so on.

Although *C++ Network Programming* covers the essentials, its style and multiple (obscure) objectives make reading laborious. They should take a page from the *Berkeley DB* book and cut to the chase. That way they might not only combine volumes 1 and 2, but also produce a book that you'd want to read in a single sitting. Buy the book if only to sweeten the bitter pill of reading the online ACE documentation.

★★★★½ | *Berkeley DB*
New Riders Publishing
www.newriders.com

★★ | *C++ Network Programming, Volume 1* | Addison-Wesley
www.addison-wesley.com

Crosbie Fitch currently gets paid to do a bit of C++ at Qube Software Ltd., but not at www.cyberspaceengineers.org.

Tetsuya Mizuguchi:
On Game Artists and Games As Art

Tetsuya Mizuguchi is the affable president and CEO of Sega's United Game Artists division. Best known previously for the popular SEGA RALLY series, Mizuguchi helped solidify the Dreamcast's reputation for uniquely stylish games with 2000's SPACE CHANNEL 5. Leaving Sega's swan-song console behind and flush with the music-action success of SPACE CHANNEL 5, Mizuguchi brought REZ to the Playstation 2 in late 2001. A musical shooter inspired by abstract expressionist Wassily Kandinsky and steeped in the sensory anarchy of synesthesia, REZ has been hailed by critics and developers alike as a milestone in artistic expression with games, but Mizuguchi maintains a distinction between games and other art forms.

United Game Artists' Tetsuya Mizuguchi, affable creator of SPACE CHANNEL 5 and REZ.

Game Developer. **What were your motivations and goals in creating REZ?**

Tetsuya Mizuguchi. The idea for REZ is something that we have been contemplating since 1995, but we could not realize it with the technology at that time. So we could do nothing but just wait for hardware to improve. The starting point was the idea that we wanted to transform the pleasure of shooting games into music. The idea was that shooting would assimilate into sounds, and your own actions would become music. My goal with REZ was to design human ambitions and desires into a game by using cutting-edge technology. I develop everything with this same motivation in mind — from SEGA RALLY to REZ to SPACE CHANNEL 5.

GD. Most developers I've talked to are quite impressed with REZ's unique gameplay and player experience. Since all creators are naturally influenced by other works they see and experience, what games influenced REZ, and what influence would you like REZ to have on the development of other games?

TM. As a general rule, I do not get inspiration from other games. Rather, I am inspired in a variety of ways as I go along my journey and talking to various people. REZ was not directly influenced by any particular games, however, the game is a conscious continuation of excellent shooting games from the past, like XENON 2 or XEVIOUS.

GD. You've spoken often of the synesthesia you try to create in your games, the melding of the senses. Do you envision and design around a certain experience you want players to have, or is it better to try to define a good framework for a range of potential experiences?

TM. I do not design a game around a particular experience, and I don't think it's practical to include so many variations of experiences into a single game. Although the core of the gameplay should focus on only one or two personal experiences, we must consider how we can meld the senses to achieve this.

GD. You have the word "artist" in the name of your studio. Do you think of REZ as a work of art?

TM. I do not regard REZ as a work of art — it is a game. Even though REZ shares some artistic aspects that have been inspired by Kandinsky, we still consider REZ entertainment, not art. I named my studio United Game Artists because we wanted every single member of staff to always be creative and innovative toward games. That is why we say "game artists," not "artists." Our name means that we want our studio to be a union of game artists.

GD. How do you orchestrate the individuals on your development team to work together to deliver a unified concept, especially one that is unique and hasn't been seen before?

TM. All my team members have totally different personalities, and this is sometimes intentional when we are recruiting our staff. Game development is a group activity, and the producer's job is to manage how to bring out everyone's special talent and to get them together into one. It is not words that help us create a unified concept, but sharing common experiences.

GD. What kind of games will you be making five years from now?

TM. It's a secret (*laughs*).

ored
278

Scalar Quantization

In last month's column, "Packing Integers," I revealed some convenient techniques for fitting things in a small space for save-games or for saving bandwidth for online games (massively multiplayer games in particular spend a lot of money on bandwidth). This month I'll extend these methods to include floating-point values. I'll do this by converting the floating-point values to integers, then combining the integers.

Let's say we want to map a continuous set of real numbers onto a set of integers, a process known as quantization. We start by splitting the set into a bunch of intervals and then assigning one integer value to each interval. When it's time to decode, we map each integer back to a scalar value that represents the interval. This scalar will not be the same as the input value, but if we're careful it will be close. So our encoding here is lossy.

Game programmers often perform this general sort of task. If they're not being cautious and thoughtful, they'll likely do something like this:

```
// 'f' is a float between 0 and 1
const int NSTEPS = 64;
int encoded = (int)(f * NSTEPS);
if (encoded > NSTEPS - 1) encoded = NSTEPS - 1;
```

Then they decode like this:

```
const float STEP_SIZE = (1.0f / NSTEPS);
float decoded = encoded * STEP_SIZE;
```

Because I want to talk about what's wrong with these two pieces of code and suggest alternatives, I'll call the first piece of code "T," for Truncate. It multiplies the input by a scaling factor, then truncates the fractional part of the result (the cast to `int` implicitly performs the truncation).

The second piece of code, which I will call "L" for Left Reconstruction, recovers a floating-point value by scaling the encoded integer value, giving us the value of the left-hand side of each interval (Figure 1a). Using these two steps together gives us the TL method of encoding and decoding real numbers.

Why TL Is Bad

As Figure 1a shows, the net result of performing the TL process on input values is to shunt them to the left-hand side of whichever interval they started in. (If you don't see this right away, just keep playing with some example input values until you get it.) This leftward motion is bad for most applications, for two main reasons: First, our values will in general shift toward zero, creating a bias toward energy loss. Second, we end up wasting storage space (or bandwidth). To see why these two problems exist, let's look at an alternative.

I am going to replace the L part of TL with a different reconstruction method, which is mostly the same except that it adds an extra half-interval size to the output value. As a result, it shunts input values to the center of the interval they started in, as Figure 1b shows. I'll call this piece of code "C," for Center Reconstruction:

```
const float STEP_SIZE = (1.0f / NSTEPS);
float decoded = (encoded + 0.5f) * STEP_SIZE;
```

When we use this code together with the same truncation encoder as before, we get the codec TC. We can see from Figure 1b that TC increases some inputs and decreases others. If we encode many random values, the average output value converges to the average input value, with no change in total energy.

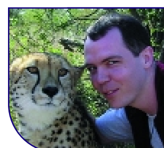
Now let's think about bandwidth. The amount of storage space used is determined by the range of integers we reserve for encoding our real-numbered inputs (the value of `NSTEPS` in the code snippets above). In order to find an appropriate value for `NSTEPS`, we need to choose a maximum error threshold by which our input can be perturbed.

When we use TL to store and retrieve arbitrary values, the mean displacement (the difference between the output and the input) is

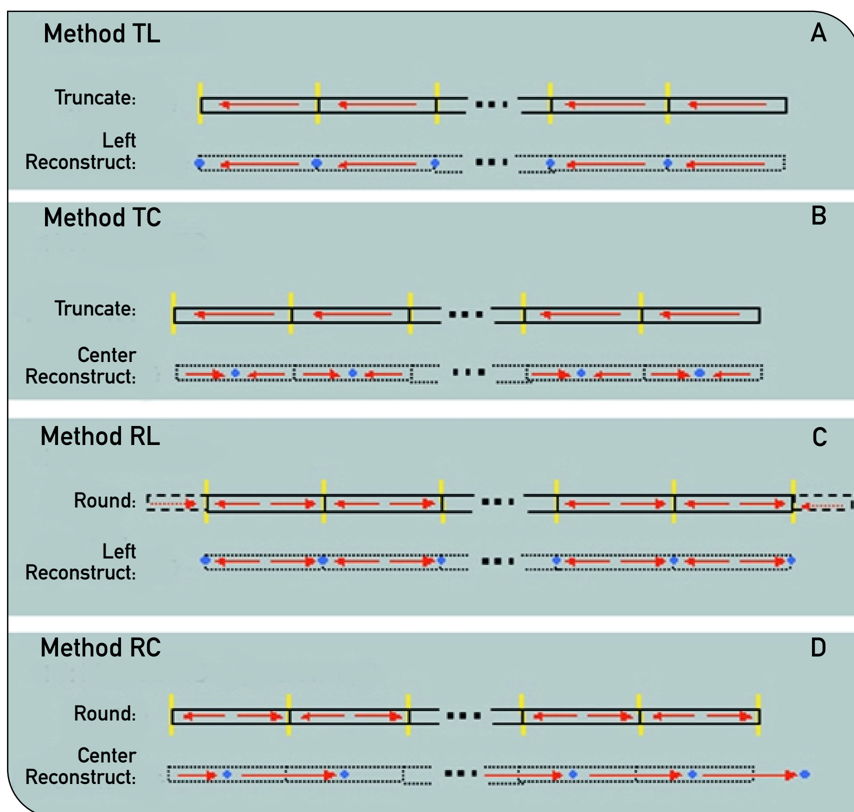
$$\frac{1}{s}\int_0^s x\,dx = \frac{1}{2}s,$$

where s is 1/`NSTEPS`.

When we use TC, the mean displacement is only $1/4s$. Thus, to meet a given error target, `NSTEPS` has to be twice as



JONATHAN BLOW | *Jonathan is a game technology consultant living in San Francisco. He is addicted to Pocky. Send him e-mail at jon@bolt-action.com.*



FIGURES 1A–1D (from top to bottom). Four methods of quantizing real numbers. The yellow notches represent the integers; the red arrows indicate which real numbers map to each integer, and the blue dots show which real numbers will be reconstructed.

high with TL as it is with TC. So TL needs to spend an extra bit of information to achieve the same guaranteed accuracy that TC gets naturally.

Why TC Can Be Bad

Though TC is a step above TL in many ways, it has a property that can be problematic: there's no way to represent zero. When you feed it zero, you get back a small positive number. If you're using this to represent an object's speed, then whenever you save and load a stationary object, you'll find it slowly creeping across the world. That's really no good.

We can fix this by going back to left reconstruction, but then, instead of truncating the input values downward, we round them to the nearest integer. We'll call this "R" for Rounding. As programmers, we know that you round a nonnegative number by adding 1/2 and then truncating it. Thus the source code is:

```
// 'f' is a float between 0 and 1
const int NSTEPS = 64;
int result = (int)(input * (NSTEPS - 1) + 0.5f);
if (result > NSTEPS - 1) result = NSTEPS - 1;
```

Figure 1c demonstrates method RL. It's got the same set of output values that TL has, but it maps different inputs to those outputs. RL has the same mean error as TC, which is good. It can store and retrieve 0 and 1; 1 is important, since, if you're storing something representing fullness (such as health or fuel), you want to be able to say that the value is at 100 percent.

It's nice to be able to represent the endpoints of our input range, but we do pay a price for that: RL is less bandwidth-efficient than TC. Note that I changed the scaling factor from `NSTEPS`, as TC uses, to `NSTEPS - 1`. This allows us to map values near the top of the input range to 1. If I hadn't done this, values near 1 would get mapped further downward than the other numbers in the input range and thus would introduce more error than we'd like. Also, RL's average error would have been higher than TC's, and it would have regained a slight tendency toward energy decrease. I avoided this situation by permitting the half-interval near 1 to map to 1.

But this costs bandwidth. Notice that the half-intervals at the low and high ends of the input range only cover one interval's worth of space, put together. So we're wasting one interval's worth of information, made of the pieces that lie outside the edges of our input. If an RL codec uses n different intervals, each interval will be the same size as it would be in a TC codec with $n - 1$ pieces. So to achieve the same error target as TC, RL needs one extra interval.

If our value of `NSTEPS` was already pretty high, then adding 1 to it is not a big deal; the extra cost is low. But if `NSTEPS` is a small number, the increment starts to be noticeable. You'll want to choose between TC and RL based on your particular situation. RL is the safest, most robust method to use by default in cases where you don't want to think too hard.

Don't Do Both

Once, when I was on a project in which we weren't thinking clearly about this stuff, we used a method RC, which both rounded and center-reconstructed. Figure 1d shows the results of this unfortunate choice. The result is arguably worse than the TL that we started with, because it generally increases energy. Whereas decreasing energy tends to damp a system stably, increasing energy tends to make systems blow up. In this partic-

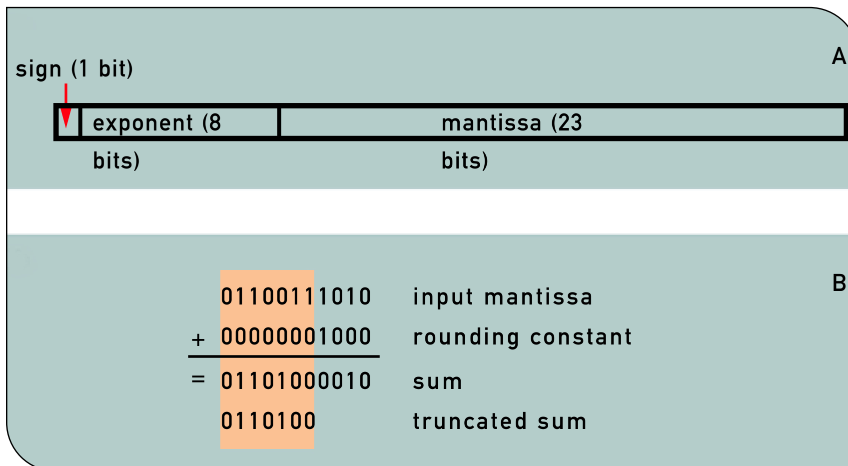


FIGURE 2A (top). The format of a 32-bit IEEE floating-point number. FIGURE 2B (bottom). Adding a constant before truncating the mantissa (the yellow-colored area) represents the number of bits to which we want to truncate.

ular project, we thought we were being careful about rounding. However, we didn't do enough observation to see that the two rounding steps canceled each other out. Live and learn.

Interval Width

So far we've been assuming that the intervals should all be the same size. It turns out that this is optimal when our input values all occur with equal probability. Since I'm not talking about probability modeling in this article, I'll just assume intervals of equal size (this approach will lend us significant clarity next month, when we deal with multidimensional quantities). But you might imagine that, if you knew most of your values landed in one spot of the input domain, it would be better to have smaller intervals there and larger ones elsewhere. You'd probably be right, depending on your application's requirements.

Varying Precision

So far, I've talked about a scheme of encoding real numbers that's applicable when we want error of constant magnitude across the range of inputs. But sometimes we want to adjust the absolute error based on the magnitude of the number. For example, in floating-point numbers, the absolute error rises as the number gets bigger. This property is useful because often what we care about is the ratio of magnitudes between the error and the original quantity. So with floating point, you can talk about numbers that are extremely large, as long as you can accept proportionally rising error magnitudes.

One way to implement such varying precision would be to split up our input range into intervals that get bigger as we move toward the right. But in the interest of expediency, I am going to adopt a more hacker-centric view here. Most computers we deal with already store numbers in IEEE floating-point formats, so a lot of work is already done for us. If we want to save those numbers into a smaller space, we can chop the IEEE representation apart, reduce the individual components, and pack them back together into something smaller.

IEEE floating-point numbers are stored in three parts: a sign bit, *s*; an exponent, *e*; and a mantissa, *m*. Figure 2a shows their layout in memory. The real number represented by a particular IEEE float is $-1^s m \times 2^e$. The main trick to remember is that the mantissa has an implicit "1" bit tacked onto its front. There are plenty of sources available for reading about IEEE floating-point, so I'll leave it at that.

If we know that we're processing only nonnegative numbers, we can do away with the sign bit. The 32-bit IEEE format provides eight bits of exponent, which is probably more than we want. And then we can take an axe to the mantissa, lowering the precision of the number.

Our compacted exponent does not have to be a power of two and does not even have to be symmetric about 0 like the IEEE's is. We can say, "I want to store exponents from –8 to 2," and use the Multiplication_Packer from last month's column to store that range. Likewise, we could chop the mantissa down to land within an arbitrary integer range. To keep things simple, though, we will restrict our mantissa to being an integer number of bits. This will make rounding easier.

To make the mantissa smaller, we round, then truncate. To round the mantissa, we want to add .5 scaled so that it lands just below the place we are going to cut (see Figure 2b). If the mantissa consists of all "1" bits, adding our rounding factor will cause a carry that percolates to the top of the mantissa. Thus we need to detect this, which we can easily do if the mantissa is being stored in its own integer (we just check the bit above the highest mantissa bit and see if it has flipped to 1). If so, we increment the exponent.

Rounding the mantissa is important. Just as in the quantization case, if we don't round the mantissa, then we need to spend an extra bit's worth of space to achieve the same accuracy level, and we have net energy decrease as well.

Sample Code

This month's sample code implements the TC and RL forms of quantization. It also contains a class called Float_Packer; you tell the Float_Packer how large a mantissa and exponent to use and whether you want a sign bit. You can load...

FOR MORE INFORMATION

Hecker, Chris. "Let's Get to the (Floating) Point." *Game Developer* (February/March 1996).

www.gdmag.com.

From Source to Texture

Last month, I looked at the process of gathering source material to use as a starting point for generating textures. This month, we'll continue with the next stage, processing these images and making them into a finished texture.

The particular style constraints that apply to an individual game's visuals vary. Making textures for a kid-friendly RAINBOW WORLD OF FLUFF will obviously take you down a different path from building a texture set for SLAUGHTERHOUSE CARNAGE TOURNAMENT. However, there are some general approaches that can be helpful to whatever style of game you're working on.

Some of the following points may seem obvious to those who have been working with textures for a while. However, since there is usually more than one way to work, it's always best to look at multiple available methods.

Powers of Two

Powers of two are the basis for most texture sizing. In mathematical terms, power of two refers to the process of multiplying the number two by itself a particular amount of times, giving us the sequence that anyone working with computers is familiar with: 2, 4, 8, 16, 32, 64, 128, 256, 512, and so on.

Sizing textures to fit within these constraints (Figure 1) reduces the amount of time the processor takes to handle a texture, thus speeding the process up considerably. A screen full of textures whose sizes are something in the order of 137×86 pixels will most likely force your engine to jump through a whole host of unnecessary hoops.

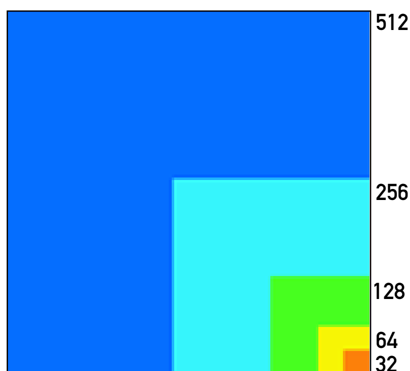


FIGURE 1. Power of two texture sizing.

Bigger Is Better

Assuming for a moment that the standard size of most textures in your game is 256×256, it may seem best to work on them at this size so that you always have an accurate idea of what the finished texture will look like. In most cases, however, working at that size is not ideal. If you have the luxury of using an extremely high resolution scan as your starting point, working on the texture at 1024×1024 is usually a bonus. There are two main reasons for this.

First, shrinking an image down often makes the features added into an image integrate better than they would at its original scale. The resampling process smoothes out small areas that may have looked a little rough. In addition, working large generally gives the artist more freedom to work quickly, because he or she knows that every pixel does not have to be perfect, and the scaling-down process will produce a more forgiving end result.

Second, the development of a game can take longer than expected, as we all know. Sometimes, as development rolls on, the hardware-induced restraints are relaxed several times. This may be a result of refinements in your engine, an increase in what is considered minimum spec for a current PC, or if you're really taking a long time, a new generation of consoles emerging with whole extra levels of performance enhancements. Whatever the reason, working on (and more importantly saving a version of) a texture at as large a size as possible protects you from the problem of finding that all your textures are saved at a smaller size than what your

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

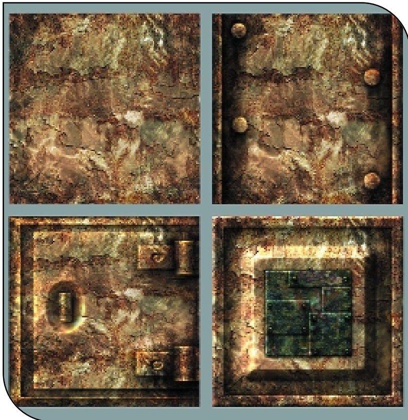


FIGURE 2. Using base layers with variations speeds workflow and helps unify the color palette.

engine can actually handle. Even Photoshop's might is unable to retrieve details that are lost from the original image when it's scaled down, and scaling up is practically pointless.

Keep the Layers

I may be presumptuous, but I suspect that most people working on textures do so in Photoshop, and if so, are familiar working with layers. So, in the same way that I recommended artists save as large a copy of each texture as possible, I also suggest maintaining a copy of each texture with all its layers intact.

The temptation to flatten and save once you are happy with the result can be quite strong. Saving the texture with its layers gives you more options for alteration at a later stage, without the time-consuming process of working an image through once again from the start.

Work from a Generic Base

If you need a texture set that must look like it fits together, you'll save time and be more effective if you establish a base starting point and then modify it to create variation. The base ought to be fairly feature free but should provide the background for the rest of the textures (Figure 2).

This approach works well with interiors and can be used in exteriors to some extent. However, the innate variation in organic forms does limit somewhat its usefulness outside.

Managing the Color Palette

Using the base layer approach goes some way toward unifying your color palette. When considering the range of colors a complete texture set will contain, artists should consider four areas: consistency, variation, harmony, and collision.

Consistency. Consistency refers to the idea that colors chosen within a specific texture set must remain constant for things that are meant to represent the same thing. For example, the color of the clumps of grass in a muddied earth texture would have to fit with the main coloration of the grass texture with which it is used in conjunction.

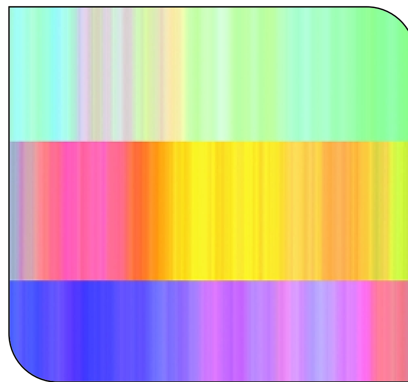


FIGURE 3. Harmonious colors help pull an environment together as a whole.

Variation. This element is the antithesis of monotony and presents the opportunity to provide visuals that impress and entice the player. Providing textures that utilize as many of the available colors as possible (without breaching the boundaries of consistency and overall styling) helps to avoid dullness in an environment.

Harmony. Presenting harmony in textures can reinforce the feeling that an environment works as a whole and is not just a grouping of isolated elements (Figure 3). Simply put, some colors work well together and some don't. There are books that can help with this (see For More Information) but as an artist you should have your own sense of what works and what doesn't.

FIGURE 4. Color collision helps alert players to something unusual about the environment, such as dangerous areas.

Collision. Collision is the opposite of harmony. In nature, colors that don't fit with the environment often signify danger of some kind, like some poisonous snakes, for example. Like highway work crews before us, we have borrowed this concept to some extent when marking out areas or objects in our game that require caution (Figure 4). For artists assembling a texture set, collision can be useful for alerting the player (even if not overtly) to the possibility of danger.

Darkening and Lightening

Adding features within a texture often requires a certain amount of darkening and lightening of areas to give the impression of depth. As with traditional painting, just adding black or white won't achieve the effect particularly well. For more flexibility, it is worth considering the blending modes of screen, multiply, dodge, and burn (Figure 5).

Screen. Selecting a screen operation multiplies the inverse of the blend (the color you are using) and the base color (the color of the image itself) in each channel. This operation results in a lighter color than is currently present in

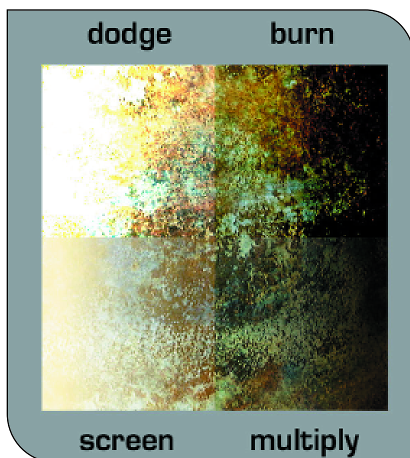


FIGURE 5. Blending modes lighten and darken textures to help create depth and other effects, but use with caution.

the image, unless you choose black as the blend color, which results in no change. Effectively, the screen operation allows you to choose the color that you'll use as a base for lightening, but when it's applied, it leaves the underlying detail intact (as opposed to simply painting over with a lighter color).

Multiply. This is essentially the opposite of screen, where the base and blend (not the inverse blend) colors are multiplied in each channel. This process produces a darker color. Here again, the underlying detail is not destroyed, just darkened. Depending on the blend color you select, you can obtain a more realistic shadowing effect.

Dodge. In the dodge technique, you use the color information in each channel to produce a brightening effect in the direction of the blend color. The basic result sounds like it would be similar to that of screen, but dodge operates more aggressively on brightness levels as well as hue, and the results are a lot stronger. Oversaturation can occur with dodge, but the technique is most useful when you need to simulate lights or metallic highlights, when there is a very rapid rise in perceived brightness.

Burn. Burn is to dodge as multiply is to screen. Burn uses the color information in each channel and darkens in the direction of the blend color. However, burn is perhaps even more dangerous when it comes to oversaturation than dodge, as its darkening effect achieves nothing in areas that are already at maximum saturation.

Contrast, Brightness, and Level Adjustments

Textures designed to be used in a game often require higher levels of contrast than what is present in the original image from which they are taken. Exaggerating the variation across a texture helps reduce the flattening effect that in-game lighting can often have and can help make features more clear (Figure 6). You also may need to raise overall brightness levels if the texture is too dark.

A basic adjustment with the brightness and contrast controls may provide the desired result, but this tool is broad in its effect, the values cannot be saved, and in some cases the effects are too generalized. If this is the case, level adjustment may be what you're after.

Adjusting levels gives you tightly focused control over each aspect of the image. In brief, the level adjustment dialog gives access to each of the color channels as well as a combined RGB channel, allowing you to adjust midtones, shadows, and highlights separately. In addition, an output level controller allows you to remap the intensity values for the entire image to correspond to any range you set. This lets you constrain the total level of contrast within new limits that you can shift away from the dark or the light end of the scale as required.

Most of the time, you'll make adjustments in the combined RGB channel (unless specific color correction is needed). Once you're happy with the settings, you can save them for later use, which is essential if you find that you need to match the adjustments you made earlier but neglected to write all the values down.

Both contrast and level adjustments also affect saturation. More often than not, you'll need to compensate for this change by reducing the overall level of saturation, so it is closer to its initial value.

FIGURE 6a (left) AND 6b (right). Adjusting the levels of the initial image (6a) makes texture features more clear and can reduce the flattening effects of in-game lighting (6b).

Harness the Power

All in all, the power of today's top image-manipulation packages allows the artist a huge amount of control when converting an image into a texture. With practice and experience, this process can become quite rapid as you become familiar with what works best and how to get the result you're looking for with the least amount of pain.

FOR MORE INFORMATION

Birren, Faber, and M.E. Chevreul. *The Principles of Harmony and Contrast of Colors and Their Applications to the Arts.* Schiffer Publishing Ltd., 1987.

Sahawata, Lisa. *Color Harmony Workbook: A Workbook and Guide to Creative Color Combinations.* Rockport Publishers, 1999.

Charting a Course for Game Audio

I've always said that the videogame industry right now is a lot like the film industry was in the 1940s and '50s. We are a young industry, one which is still feeling out and incorporating new standards and methods each day — not only in the creative arena, but in technical and business aspects as well. Being in audio for videogames over the last 11 years, I've heard all of the concerns, complaints, and industry criticisms from my fellow audio compadres.

In February 2002, several leading audio professionals in the game industry got together to form an organization focused on promoting excellence in interactive audio, and we officially launched the Game Audio Network Guild (G.A.N.G.) at GDC in March. G.A.N.G. is a nonprofit organization established to raise awareness of interactive audio by providing education, information, instruction, resources, publicity, guidance, community, recognition, and enlightenment not only to its members, but also to content providers and listeners throughout the world. G.A.N.G. aims to advance the gaming industry by helping produce more competitive and entertaining products, while supporting career development and education for aspiring game audio professionals, publishers, developers, and students.

There is no evil plot purposely trying to stomp on game audio. Developers and publishers don't secretly wish that their audio will suck. The fact is that a lot of times it's just a matter of educating the masses, learning and sharing from our past experiences, banding together collectively as a group, and acknowledging what needs to be fixed and addressing the issues head-on. Why is it that some of the best and most popular games we've ever played fell short in the audio department? Why are the dialogue and acting in some very big-selling games out there so awful and unbearable? Anyone else sick of that one single footstep sound you too often hear throughout an entire game? Not only do we need to educate ourselves, game audio professionals also need to teach the up-and-comers as well as supply information and resources to professionals from other industries, such as film, television, and music, so that they too can accomplish great-sounding audio for games.

This broad approach is why G.A.N.G. is not meant to be a boys' club for a select few. In addition to its other goals and initiatives, G.A.N.G. is establishing a variety of programs including the annual G.A.N.G. Awards, the G.A.N.G. Seal of Approval, and various membership levels which will all help promote recognition and audio quality to the game industry.

Let's say that a publisher or developer is thinking about putting live orchestra into their game. Where do they go? Whom do they contact? What is the cost? What services are available? How many instruments would they need? What are the benefits? What are the possible production pitfalls? Where can they hear each of the orchestras they're considering? G.A.N.G. aims to provide answers to these questions and other quandaries. But G.A.N.G. isn't just about audio people, it can educate the entire gaming industry. Perhaps once developers see how well G.A.N.G. works for everyone, other game makers will form their own specific guilds: artists, programmers, designers, producers, and even testers could benefit greatly from a sharing of their unique knowledge and experience.

All Together Now

Companies such as Dolby and DTS have already committed sponsorship, time, and marketing resources to the organization. In addition, all of the major audio manufacturers are getting behind G.A.N.G. because our goal is their goal: improving audio. We have representatives from Microsoft, Sony, and EA who sit on our board of directors or who are G.A.N.G. advisors and summit attendees.

Our mission statement and member benefits are very ambitious, and we realize it may take years to complete our goals. But we have to start sometime with something. We are willing to start now and put all of the time, energy, resources, money, and knowledge into something we all believe will benefit not only ourselves and the gaming industry today, but also the future generations of game makers. In the meantime, we're very interested to hear from all videogame professionals to find out how a professional guild such as G.A.N.G. can help them as creators refine and advance their art.

TOMMY TALLARICO | *Aside from being the president of Tommy Tallarico Studios Inc., Tommy is the host and co-producer of the* Electric Playground *and* Judgment Day *television programs. He is also the acting president and founder of G.A.N.G. You can contact Tommy at tallarico@aol.com. For more information on G.A.N.G., see the G.A.N.G. web site at www.audiogang.org.*

Game Design at GDC 2002

Instead of another rule, this month I'm doing something a little different. I'm writing this column just days after the conclusion of the 2002 Game Developers Conference, and several developments there bear on The 400 Project and the idea of design rules in general.

Hal Barwood and I gave a lecture this year called "More of the 400: Discovering Design Rules," which covered the concept behind the project and made the case for design rules as a good tool for game designers to improve the state of their craft. We believe that design is a planning process, where one proceeds from murk to clarity, successively improving and refining a concept. But the process is not a mechanical or deterministic one, and it requires knowledge of not only games and production methodology, but also a keen appreciation of human nature and a sense of what is fun.

It's essential that the human element be considered and so we take a linguistic approach, in part to avoid potentially rigid software-engineering techniques as the template for game design. Some rules we propose can be bent, others can be broken — but having a conscious appreciation of what those rules are is an important prerequisite to being able to bend or break them and move toward order, not chaos. Rules are tools that provide instructions to the designer, not just observations on the nature of what has been done previously. To be most useful, they must be reasonably concrete and aimed at practical use, not pure academic discourse.

One essential part of the format of these rules is the concept of trumping, or documenting how the rules affect each other and showing how to determine precedence when they conflict. This is a tough chicken-and-egg problem, as it's hard to propose rules that contain trumping information when there is no existing list of proposed rules. But as regular readers of this column have seen, we're going ahead anyway, proposing trumping information based on the growing base of rules, and using common sense where the lack of comparable rules leaves a gap. If you've been thinking of submitting a rule but are stymied by the problem of determining trumps, don't hesitate — better that we reach critical mass first, and then we go back and find the interconnections.

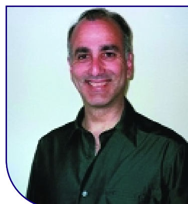
At this year's GDC, we recapped the four rules Hal proposed last year (the original "Four of the 400" talk at GDC 2001) and explained six new ones, some of which have been published in previous installments of this column. Some of the new ones include "Emphasize Exploration and Discovery," "Let Players Turn the Game Off," and "Build Subgames." We found an increasing number of rule overlap and conflict, confirming the usefulness of focusing on trumping information.

The quest for better ways to discuss and design games was a common theme at this year's GDC. Other methods proposed included design heuristics, a design grammar, and several different approaches using design patterns based on architect Christopher Alexander's works. We had some fascinating discussions both in the formal and informal venues of lectures, roundtables, hallway conversations, and shop talk at parties. It's far beyond the scope of this column even to summarize the wide range of ideas that were proposed and debated, but I found particular inspiration in Bernd Kreimeier's roundtable discussions on design patterns. It was promising to see the amount of agreement between the different approaches, and I expect we'll see some convergence in the future. I'll continue to report on it in this column.

To conclude with a related anecdote from GDC, I had the pleasure of attending the second of Steve Meretzky's roundtables on game addictiveness. He had collected a number of suggestions and observations about ways that games have been made addictive in the past, many of which would make good rules. He also quoted some research on changes in brain activity when people play games, in particular referring to an article called "This Is Your Brain on TETRIS" (*Wired*, May 1994).

TETRIS was often cited (along with CIVILIZATION in its various incarnations) as a primary example of addictive game design. In a moment reminiscent of the scene in *Annie Hall* where Woody Allen hears someone in line ahead of him pontificating on Marshall McLuhan, and then produces McLuhan from behind a corner to refute the claim, Alexey Pajitnov himself appeared in midsession and defended his conscious decisions to make TETRIS addictive; he contended that the best gameplay can be real quality time.

We're still a young enough industry that we have to strive to define and codify the very techniques we use to create games — but that very same youth means that many of the trailblazers of our industry are still available to reveal their own thoughts and insights about their work. I invite you to be a trailblazer yourself and send in your own rules or thoughts on this process to me.



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

Game Development Myth vs. Method

As technology and production values have changed, there's been a rise of nostalgia in our industry for the days of the garage developer. Part of this is real nostalgia, but in other ways it masks a very legitimate fear — the fear that either our industry will become a cynical, uncreative corporate animal, or that it will become financially untenable.

As developers, we share this concern too. Nevertheless, having had a unique opportunity in our nearly 30 years of combined experience to observe the good and bad in game development, we believe that it's possible to build games in a fashion that not only fosters the necessary creativity to create an exciting game, but also goes a long way toward alleviating the corporate risk that threatens to torpedo creativity along the way.

For lack of a better name, we've called our strategy "Method." Our Method is built on four keystones, which we'll describe one by one: the distinction between preproduction and production, the "publishable" first playable, macro versus micro design, and gameplay testing.

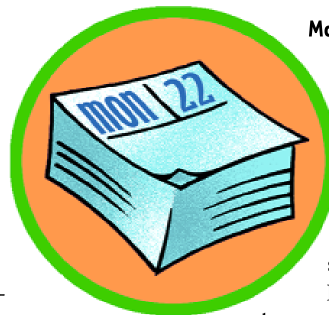
Preproduction vs. Production

With the notable exception of highly sequelized or serialized games, the beginning of every project is primarily concerned with looking for a lightning strike of inspiration. Just as a musician uses a piano to compose a song, or a painter begins with sketches and studies, game preproduction is about creating a canvas on which to find the core concept or feature that will set the game apart.

Preproduction is hard. In fact, preproduction is so much harder than production that lots of teams just skip it, or give it short shrift and head on into production. Our guess is that 80 percent of mistakes in game development are the direct result of things done — or not done — in preproduction.

These mistakes often befall those who believe in one of what we have identified as seven myths in game development:

MYTH #1: PLANNING
"It's possible to plan and schedule the creation of your game."



Managing chaos. The division of preproduction and production is as much a strategic one as it is a creative one. Preproduction is certainly hard for the creative people who are trying to invent the game, but it's also hard for the management assigned to shepherd the process along.

Planning is hard in preproduction because it must be allowed to be a chaotic process. You cannot plan when inspiration will strike, nor can you schedule the date when you will have worked out all your seemingly intractable problems. It's not just a bad habit — it's impossible.

So if your schedules and charts are useless in preproduction, how do you manage chaos?

First, assemble a core team. When faced with a chaotic situation, your inclination should be to find your smartest, most experienced (and probably highest-paid) staff to comprise your core team. This small core team (perhaps as few as four or five people) will determine everything that's important about your game, and they will most likely go on to become your team leaders during production.

Unfortunately, it's a common habit to kick off the preproduction phase using junior people who are only available because you can pull them off another project or because you just hired them. In such a scenario you are entrusting your most valued possession, your original game concept, to your least experienced employees.

With your core team in place, the team must create successive prototypes. It's important not to wait before you start making a prototype. Take the pieces you have, however sketchy, and build the best you can. Prototypes are where you learn.

The first prototypes will by necessity be simple and limited in their ambition; however, the prototypes should become indistinguishable from game levels relatively quickly. Each of these "real-level" prototypes brings together artwork, game mechanics, and technology to show what could be an entire level of

MARK CERNY & MICHAEL JOHN | *Mark Cerny has been working in the videogame industry for over 20 years, on games ranging from Marble Madness to Crash Bandicoot and Jak & Daxter. Michael John is a 10-year veteran of the game business, with credits including the Spyro the Dragon series. Mark and Michael have been working together since 1995 and are currently doing contract design and production work for a number of clients as Cerny Games, Inc.*

your game, if you were to finish it.

Which you won't. Which brings up another myth:

MYTH #2: PRODUCTIVITY
"Working productively means not throwing out good work."

At Cerny Games, we plan on five of these real-level prototypes. This means that we discard four completed levels of our game. Using the character-action genre as an example, these games tend to have about 20 levels when completed — so we're talking about throwing out 20 percent of a game.

In fact, throughout preproduction you will not create any material designed to be played by the public. If you are very lucky, you might find a use for the best of your preproduction work.

But don't count on it, and for God's sake don't schedule for it.

As a matter of fact, that's the second part of the definition of preproduction.

Building a game design. In preproduction, it's essential to remember that you're not making a game — not yet. This sounds frustrating and difficult for developers, and to managers it sounds half-assed and impossible to schedule. All of this is true. So what are you doing during preproduction? Through your real-level prototypes, you're actually building your game design,

but not your actual game.

Game design at this stage should include at least the following things:

- The three Cs: character, camera, and control. (The three Cs we employ are somewhat specific to a character-action game, but there are analogues in every genre. For instance, you might replace "control" with "interface" in a real-time strategy game, or "character" with "car" in a driving game.)
- Your game's look.
- Completed key technology.

Looking at this list, you can see why preproduction is so difficult. Why then is all this so important to achieve before building the first level of the published product?

In order to get a game design worth building, you have to take chances. You have to make best guesses at how your game will work and try it out in as real a setting as you can put together.

Taking chances during preproduction means finding yourself, for example, building levels without complete knowledge of your character's move set, without knowing the real limits of your technology, or without knowing the global context of the piece of the game you're working on.

These are all absolutely terrible things to be doing during production and should be considered signs that your game is in serious trouble. In preproduction, however, these are exactly the kinds of things you want to be doing.

From a financial standpoint (which is also a metaphor for what you're doing inside the development team), this is about spending money now to save money later. If you've ever cancelled (or thought about cancelling) a project near completion, wouldn't you rather have done five prototype levels before fully funding it?

It's these kinds of managerial and financial concerns that lead us to our next myth:

MYTH #3: MILESTONES
"Frequent project review is essential to good management."

Milestones are wonderful during production; they allow you to break down the game into manageable chunks and set deadlines to track your progress. Having said that, milestones should not exist during preproduction.

Preproduction is chaotic, somewhat disorganized, and doesn't produce predictable results. And yet we often see and hear of expectations for teams in preproduction to produce viewable, playable milestones for internal or external review.

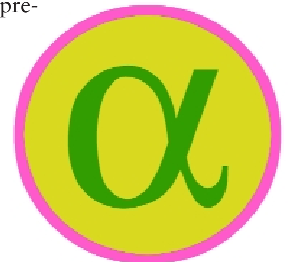
Once a game's in full production, it's not too tricky to bring all aspects of the game together for a milestone. In preproduction, on the other hand, not only will a milestone have little resemblance to the real product, milestones do significant damage to the preproduction process. In preproduction, the effort put into generating a viewable milestone is expressly subtracted from effort that could have been put into preproduction of the game itself.

Simply put, a team in preproduction should be considered "offline" when it comes to formal deliverables. This situation puts managers in a quandary: How, without formal product reviews, can preproduction be responsibly managed?

As a start, adhering to the strict discipline of a publishable first playable keeps pressure in the right place on a preproduction team. In addition, there can be a prenegotiated term for preproduction, a limit on how long it can continue. If the production team is not making progress, continued experimentation doesn't benefit the team, the game, or the publisher.

Making experimentation results visible is another way of managing preproduction. Although traditional milestones can't be scheduled, the team will be in a rapid prototyping cycle, with all the parts for a brand-new experiment being assembled and demonstrable every month or two, ready to be shown to responsible parties in a semi-formal setting. Timing of these builds of the game must be determined by the natural flow of the work, not by an external schedule; this is a critical point in reviewing experiments.

Achieving such a level of trust between publisher and developer or management and an internal team is not easy. Management must understand that rough results — and failures — are a natural part of the preproduction experience, and teams must understand that publishers are going out on a limb when they support development without deliverables.



Ready (or Not): First Playable

Two deliverables mark the end of preproduction: a macro design document (discussed a bit later) and a "publishable" first playable.

The definition of first playable can be slippery at best, and is often abused. Sometimes it is designated as complete much too early; other times it's postponed seemingly indefinitely. In both cases the effect is the same:

MYTH #4: ALPHA = FIRST PLAYABLE

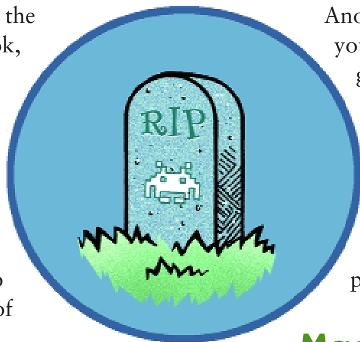
Failing to define the game fully before real levels are built is among the worst things a team can do. Taking a "What, me worry?" attitude toward first playable can lead to a game that is not defined all the way up to alpha (we've seen this disaster scenario with our own eyes).

To avert this situation, we define first playable as publishable. It's when you can look at your playable game and say, "I know exactly what this game is, and I know exactly how I'm going to build it, and it's really, really good."

So how do you determine whether your first playable deliverable is "publishable"? First, it should have at least two levels,

and those levels should demonstrate the variety that will be present in the game. Very few games can succeed without an alluring sense of variety; your first playable should demonstrate how your game will clear this hurdle.

Second, the first playable should pass a number of sniff-tests to show that your game is sufficiently compelling to be marketable 12 to 18 months in the future, when it's completed. Your look, your gameplay, and your technology all must be of such quality that an uninitiated consumer will look at it and be impressed enough to believe that it's a level from a commercial product.



If you think you have a first playable in hand, here's a checklist to see if you have two (or more) levels of publishable quality:

- Player behavior fully defined
- Basic technology done
- Enemy/obstacle behavior fully defined
- Art direction in place
- All local features included, with global features included as required
- A touch of variety
- Scope of game defined.

One of the great benefits of a rigorous first playable is that constructing a project schedule from it becomes almost trivial. Having constructed a couple of "real" game level prototypes, you will have gained a very good idea of how long it takes to put one together. The first playable also gives a firm idea of what the scope of the game will need to be.

Finally (and this is a bit of a religion with us as you'll see later in the "Game Testing" section), first playable represents your first opportunity to put the game in front of your consumers in a gameplay-test context. Do this. Note in detail how your players pick up the controls, whether they struggle with the camera, whether it's too easy or too hard, and whether they find it compelling. If something is seriously wrong, you will know. And if anything is wrong, now is the time to fix it, not later.

A time to kill. All of this discussion of first playable presumes that your game is coming along well. That may not be true. What if, after months of effort, you realize you're not going to get to the bottom of that checklist? Or perhaps you did and the game bombed in the hands of consumers. This brings us to our next myth:

MYTH #5: KILLING GAMES IS BAD
"A cancelled project is a sign of bad management or a bad team."

Actually, sometimes a cancelled project is something you should be proud of. Regardless of the talent of the team, if you can't reach a compelling first playable, it's time to kill the project and move on. You just saved several million dollars, and perhaps more importantly, a year of a good team's lives.

But how do you know when to stop a project? Unlikely as it sounds, it's very common to discover that the team simply cannot assemble everything required for a publishable first playable. If so, cancel, because if you can't get past the logistics of assembling two polished levels, you certainly can't make an entire game.

Another cancellation scenario is that you diligently create your publishable first playable and discover that you have a game not worth publishing. Canceling a project at this point is a very hard call, especially due to the emotional investment of all involved. But do it anyway. This harkens back to a central thesis underlying Method: that your project will not miraculously get better during production. You must compare your first playable with published products in your category, and your first playable has to be better than those are.

Macro vs. Micro Design

Splitting the design into two components, the macro and the micro, is the third keystone of Method. The macro design is completed by the end of preproduction, whereas the micro design is created during production.

This methodology is the result of one of the most dangerous myths in game development:

MYTH #6: DESIGN DOCUMENTS
"The more defined your initial vision, the better."

This myth sometimes takes the more insidious form of "I need a 100-page document describing my game."

Forget that. Not only do you not need a 100-page document to start your game, you don't need such a document ever.

Cerny Games has a reputation in certain circles as being anti-documentation. We consider this a badge of honor. Sitting down and writing a 100-page design document is the worst thing you can do to kick off your preproduction.

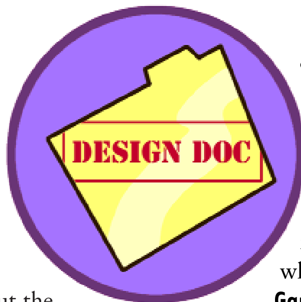
There are myriad reasons not to start with this document, but here are a few favorites:

- It's a waste of resources (including trees).
- It's deceptive — you appear to have a far more evolved idea than you could possibly have.
- You risk setting direction prior to real-world establishment of gameplay fundamentals.
- And most importantly, to quote Masaya Matsuura, creator of PARAPPA THE RAPPER: "Anything I could just write down on a piece of paper couldn't be fun!"

Macro design. Once you've finished preproduction, however, you should have a macro design document; in our Method, this document runs about five pages and includes (using a character-action game as an example):

- The character and move set
- Any exotic mechanics planned
- Description of level structure, size, and count
- Level contents

- Overall structure (linear, hub, and so on)
- The "macro chart," a one-page chart that shows all the game's dependencies and distribution of various mechanics and gameplay elements.



This document will be your roadmap for the production phase of development. It serves as the basis of scheduling for the production phase and keeps the game consistent throughout the time that it will take to finish the game.

There are a few more things that may be appropriate to have at the end of preproduction, such as story or conceptual art. These should be considered addenda and do not change the basic structure of the macro design document.

Micro design. On the flip side of development is the micro design. Micro design is strongly distinguished from macro design in that it's not represented by a document. Rather, micro design is the day-to-day work of the designers during production.

Micro design, which includes level maps, enemy descriptions, mini games, and the like, is best done on the fly. Why? Two reasons: First, if you believe that the micro design must wait for the completion of preproduction, then creating documentation at that point would mean the ridiculous act of putting the rest of the team on ice while the designers create documentation. Not likely.

More importantly, though, even after completing preproduction, you will continue to learn things during production — certain techniques, cameras, and gameplay types that work better than others. So long as you have a solid macro design and follow it, you can advance the state of the art for your game during production with confidence that you will not break the continuity or consistency of the experience.

Gameplay Testing

Let's go on to the final keystone of Method, gameplay testing, and its corresponding myth:

MYTH #7: THE CONSUMER IS KING
"If you want to make a hit, listen to the consumer."

If you want to find out what features to put in your game, or what type of game you should make, the last thing you should do is conduct a focus test.

We can sometimes be cavalier about our terminology when we discuss focus tests and gameplay tests. In our Method, we shun focus tests like week-old bread; meanwhile we attend to gameplay tests as we would to gospel.

Focus tests: Learn to say no. Humans are pack animals. If you don't believe this, then you haven't been to a moderated focus test. Focus tests inevitably devolve into popularity contests — popularity among those in the room and also popularity of ideas currently out there in the culture. So, here's a quick list of all the things you'll learn in a focus test:

- What's popular as of about 10 minutes ago
- How not to stand out
- The feature list of every game that was pretty good.

Are focus tests completely useless? Absolutely not. However, we believe in one very simple principle regarding focus tests: A focus test can only tell you what not to do. Sometimes, that's incredibly useful.

Gameplay testing: Your most vital feedback. While we may not be great fans of focus testing, gameplay testing with consumers is the last of our four keystones of Method. No game should be released without having been formally and extensively gameplay-tested during at least two points in the development process, and perhaps four or five.

Gameplay testing is simply putting your game in front of consumers — mostly the same consumers to whom you expect to sell your game — and watching them play. We may not trust what consumers say in focus testing, but we trust completely what they do in gameplay testing.

Gameplay testing should also be analyzed quantitatively as well as subjectively. You need to be able to derive statistics from your gameplay tests that allow you to tune your gameplay with a very high degree of precision. At the same time, however, it's best not to get too enamored with your statistics. What you believe represents fun is almost always wrong. That's why it's so important to be in the same room with the game players as they do their test. You can learn a lot from body language, and even though your compiled statistics can show how many times the players died, they can't tell you whether the players were enjoying themselves or hating life.

The Process Is Part of the Product

Is it possible to follow all the principles of Method on a game project? Yes. However, it requires both the publisher and the developer to commit to these principles. The publisher must be willing to accept a high degree of uncertainty and trust during the preproduction phase and exercise a decisive will to finish or start over when preproduction comes to a close.

The developers must hold up their end of the bargain too. They must commit to creating a meaningful first playable, to being transparent during their chaotic preproduction, and to submitting to time-consuming (and somewhat scary) gameplay testing multiple times during production.

That's our Method. It's a system we've identified and refined over years of making games. Is it a guarantee of success? Not even close. Games are too unpredictable and too dependent on creativity and inspiration, but we wouldn't have it any other way.

Creating a C++ Scripting System

Most of today's games require some kind of system to allow the game designers to program the story of the game, as well as any functionality that is not general enough to be directly supported by the AI or some other system. Often, the scripting system of a game is also made available to players for customization. Today it is not uncommon for third-party companies to customize an existing game to incorporate completely different gameplay, in part by changing the game scripts.

Depending on the game type, the best results are achieved using different approaches to scripting. In environments that have very well defined rules, such as RTS games, the most important task of the designer is to achieve good balance between the different units and resources, while also producing interesting maps with features that clever players can use to their advantage. Clearly, the scripting support for such games is focused on fast and easy tweaking of the different parameters exposed by the game engine, and is usually directly supported by the map editor.

Other games, for example many FPS titles, require very limited levels of customization. This is usually done by tagging objects in the game editor.

And finally, many games are driven by complex enough logic to require a complete programming language for scripting. In some cases, developers have designed and implemented their own programming languages to serve this need. To cut development time, today most game companies opt to use existing programming languages customized for their own scripting needs.

Scripting Languages

Most so-called scripting languages have one thing in common: they have been developed by a small team or even a single person for the purpose of writing simple programs to solve a particular, limited set of problems. By their very nature they are not universal, yet many end up being employed beyond their intended purpose.

Sometimes, "scripting language" simply means a programming language used to write scripts. Indeed, people have successfully used languages such as Java for scripting.

When selecting a scripting language for a game, chances are that a developer will not find a language that matches the desired functionality completely. At the very least, a developer will have to design and implement an interface between the game engine and the scripting language. The programming language the developer chooses is just one of the components — often not the most important one — of a scripting engine.

Using C++ for Scripting

One of the most important characteristics of C++ is its diversity. Unlike many other languages that are efficient for a particular programming style, C++ directly supports several different programming techniques. It's like a bag of tricks that allows many, often very different solutions to a problem. Some of the C++ features — such as function and class templates — are so powerful that even the people who developed and standardized them could not have foreseen the full spectrum of problems they can solve.

The idea of using C++ for scripting may seem strange at first. Indeed, most people associate C++ with pointers and dynamic memory management, which are powerful features but are more complex to work with than what most game designers would consider friendly.

On the other hand, C++ is the natural choice to program the rest of the game in. If the scripts are also written in C++, then integration with the rest of the code is seamless. In addition, C++ is translated to highly optimized machine code. While speed is rarely a problem for most game scripts, faster is always better.

Naturally, using C++ for scripting has some drawbacks. Because it is a compiled (as opposed to interpreted) language, a C++ program can't be changed on the fly, which is important in some applications. Also, C++ does not provide a standard for plugging in program modules at run time, which makes it nearly impossible to expose the script for customization by users (usually this is not an issue for console titles).

It's important to create a safe and easy-to-use environment for our scripts. The language used is a secondary concern, because by their very nature scripts are simple and mostly linearly executed, with limited use of if-then-else or switch state-

EMIL DOTCHEVSKI | *Emil is currently co-lead programmer at Tremor Entertainment on a soon-to-be-announced original Xbox title. His previous work includes* RAILROAD TYCOON II *for Playstation and an enhanced 3D version for Dreamcast. Emil has an M.S. in computer science from the Sofia University, Bulgaria. He can be reached at emil@tremor.net.*

ments. In this article we will demonstrate how to use some advanced C++ features to create a safe "sandbox" environment for the scripts within our game code.

Creating a Safe Scripting Environment

More often than not, programmers design a class hierarchy to organize the objects that exist in the game's digital reality. For the purposes of this article, let us assume that our game uses the classes whose partial declaration is given in Listing 1.

LISTING 1. An example class hierarchy.

```
class CRoot {
   virtual ~CRoot();
};

class CActor: public CRoot {
public:
   bool HasWeapon() const;
   bool HasArmor() const;
   int GetHealth() const;
   void SetHealth( int health );
   void Attack();
   ...
};

class CGrunt: public CActor {
public:
   ...
};

class CAgent: public CActor {
public:
   ...
};
```

Using C++ for scripting allows us to use plain pointers for interfacing with the script code. The main drawback of using pointers is that they can point to invalid memory. The benefits are that pointers are directly supported by the language and are very efficient. Also, pointers make it easy for programmers to implement and later extend the interface between the scripts and the game.

To eliminate the possibility for the script code to access invalid memory, the use of pointers must be hidden from the scripts. In C++, we can do this by organizing the pointers in a set container. Then we can define functions and operations for working with entire sets of (pointers to) objects. This neatly unifies the processing of one or more objects and usually allows the scripts to not have to handle empty sets as a special case.

For example, to represent a set of `CGrunt` objects (see Listing 1), we can use something like this:

```
std::set<CGrunt*> grunts;
```

Let's also define a functor to expose the `CActor::Attack` function to the script:

```
struct Attack {
   void operator()( CActor* pObj ) const {
      pObj->Attack();
   }
};
```

Now we can use `std::for_each` with the function object `Attack` to have all the `grunts` in the set use their attack functionality:

```
std::for_each( grunts.begin(), grunts.end(), Attack() );
```

The `for_each` template function is defined so that it can work with any sequence of objects, which is a level of flexibility we do not need. Instead, we can define our own version of `for_each`, which for convenience we can simply call `X` (from Execute):

```
template <class Set,class Functor>
void X( const Set& set, Functor f ) {
   for( typename Set::const_iterator i=set.begin(); i!=set.end(); ++i )
      f(*i);
}
```

Now we can simply say:

```
X( grunts, Attack() );
```

It's also possible to write functors that take arguments and pass them to the function they call:

```
struct SetHealth {
   int health;
   SetHealth( int h ): health(h) {
   }
   void operator()( CActor* pObj ) const {
      pObj->SetHealth(health);
   }
};
```

Now we can use the SetHealth functor like so:

```
X( grunts, SetHealth(5) );
```

Predicates

Using function objects to perform actions on an entire set of objects is a powerful feature by itself, but it becomes even more powerful if we define another version of the X function that allows us to call the function object only for selected objects in a set:

```
template <class Set,class Pred,class Functor>
void X( const Set& set, Pred p, Functor f ) {
   for( typename Set::const_iterator i=set.begin(); i!=set.end(); ++i )
      if( p(*i) )
         f(*i);
}
```

A predicate is a special type of function object that checks a given condition. For example, we can define the following predicate:

```
struct HasArmor {
   bool operator()( const CActor* pObj ) const {
      return pObj->HasArmor();
   }
};
```

Now we can write something like:

```
X( grunts, HasArmor(), Attack() );
```

This will execute the Attack functor on the members of the grunts set that are armored. Again, exposing the armor property of objects of class CActor to the script is as easy as writing a simple predicate.

Type Predicates

In the preceding examples, because Attack::operator() takes CActor* as an argument, the Attack functor can only be used with sets of objects of class CActor. Because any object of class CGrunt is also of class CActor, we can use the Attack functor with a set of grunts too. But what if we have a set of objects of class CRoot? It would be nice to be able to select only the objects of class CActor and execute Attack on them.

To do this, we need our predicates to define an output_type. Then we can design our system so that if an object passes a predicate, we can assume it is of class output_type that the predicate defines. For example, we could use the predicate IsActor that checks if a given object is of class CActor:

```
struct IsActor {
   typedef CActor output_type;
   bool operator()( const CRoot* pObj ) const {
      return 0!=dynamic_cast<const CActor*>(pObj);
   }
};
```

Note in this case that some compilers do not implement dynamic_cast efficiently because it has to work in nontrivial cases such as multiple inheritance and the like. Instead of dynamic_cast, we could use a virtual member function to do our type checks.

We also need to modify the predicate version of our X template function:

```
template <class Set,class Pred,class Functor>
void X( const Set& set, Pred p, Functor f ) {
   for( typename Set::const_iterator i=set.begin(); i!=set.end(); ++i )
      if( p(*i) )
         f( static_cast<typename Pred::output_type*>(*i) );
}
```

The output_type defined by our predicates makes it safe for the X template function to use a static_cast when calling the functor.

Now, if we have a set of objects of class CRoot, we can write:

```
X( objects, IsActor(), Attack() );
```

Complex Predicates

So now we have seen how to use predicates to execute a functor on selected objects from a given set of objects. But what if we want to combine multiple predicates to select the objects we need?

Generally speaking, it's easy to combine multiple simple predicates in a single complex predicate that our X function can check. As an example, let's define a predicate called pred_or:

```
template <class Pred1,class Pred2>
struct pred_or {
   Pred1 pred1;
   Pred2 pred2;
   pred_or( Pred1 p1, Pred2 p2 ): pred1(p1),pred2(p2) {
   }
   bool operator()( const CActor* pObj ) const {
      return pred1(pObj) || pred2(pObj);
   }
};
```

To make it possible to use pred_or without having to explicitly provide template arguments, we can define the following helper function template:

```
template <class Pred1,class Pred2>
pred_or<Pred1,Pred2> Or( Pred1 p1, Pred2 p2 ) {
   return pred_or<Pred1,Pred2>(p1,p2);
}
```

LISTING 2. Defining `pred_or`.

```
template <class Pred1,class Pred2>
struct pred_or {
   typedef typename select_child<
      typename Pred1::input_type,
      typename Pred2::input_type>::type
         input_type;
   typedef typename select_root<
      typename Pred1::output_type,
      typename Pred2::output_type>::type
         output_type;
   Pred1 pred1;
   Pred2 pred2;
   pred_or( Pred1 p1, Pred2 p2 ): pred1(p1),pred2(p2) {
   }
   bool operator()( const input_type* pObj ) const {
      return pred1(pObj) || pred2(pObj);
   }
};
```

LISTING 3. Defining `pred_and`.

```
template <class Pred1,class Pred2>
struct pred_and {
   typedef typename Pred1::input_type
         input_type;
   typedef typename select_child<
      typename Pred1::output_type,
      typename Pred2::output_type>::type
         output_type;
   Pred1 pred1;
   Pred2 pred2;
   pred_and( Pred1 p1, Pred2 p2 ): pred1(p1),pred2(p2) {
   }
   bool operator()( const input_type* pObj ) const {
      return pred1(pObj) &&
         pred2(static_cast<const typename
Pred1::output_type*>(pObj));
   }
};
```

With the `Or` function template in place, we can use `pred_or` to combine the `HasArmor` and the `HasWeapon` predicates:

```
X( grunts, Or(HasArmor(),HasWeapon()), Attack() );
```

But wait, why did we define `pred_or::operator()` to take `CActor*`? This is not ideal, because we want `pred_or` to be able to combine predicates that take objects of different classes. In addition, our `X` function requires us to define an `output_type`. What is the `output_type` for `pred_or`?

Let's extend our system to require that the predicates define not only `output_type` but also `input_type`, which is the class of objects the predicate can be checked for. With this in mind, let's define `pred_or` as shown in Listing 2.

For this to work, we need two helper template classes, `select_child` and `select_root`. We see how they work later, but for now let's just assume the following: `select_root<T,U>::type` is defined as the first class in the class hierarchy that is common parent of both `T` and `U`, or as void if `T` and `U` are unrelated. For example, `select_root<CGrunt,CAgent>::type` will be defined as `CActor`.

`select_child<T,U>::type` is defined as `T` if `T` is a (indirect) child class of `U`, as `U` if `U` is a (indirect) child class of `T`, or as void otherwise. For example, `select_child<CRoot,CGrunt>::type is` defined as `CGrunt`, while `select_child<CGrunt,CAgent>::type` is defined as void.

Indeed, for `pred_or::operator()` to return true, either the first or the second predicate should have returned true. Since we do not know which predicate returned true, our `output_type` is the root class of the `output_types` defined by the `Pred1` and `Pred2` predicates.

Similarly, because `Pred1::operator()` takes objects of class `Pred1::input_type`, and `Pred2::operator()` takes objects of class `Pred2::input_type, pred_or::operator()` must take objects that are of both class `Pred1::input_type` and class `Pred2::input_type`. This is why we need `select_child`.

Now let's define `pred_and` as shown in Listing 3. Here, the `static_cast` is justified because C++ always evaluates the left side of `operator&&` first, and then evaluates the right side only if the left side was true — and we know that if an object passes `Pred1`, it is of class `Pred1::output_type`.

Let's extend the definition of `HasArmor` to define `input_type` and `output_type` as required:

```
struct HasArmor {
   typedef CActor input_type;
   typedef input_type output_type;
   bool operator()( const input_type* pObj ) const {
      return pObj->HasArmor();
   }
};
```

Now, if we have a set of objects of class `CRoot`, we can do something like this (assuming we have defined a function template `And` similar to the function template `Or`):

```
X( objects, And(IsActor(),HasArmor()), Attack() );
```

We can even combine `pred_and` and `pred_or` in an even more complex predicate expression:

```
X( objects, And(IsActor(),Or(HasArmor(),HasWeapon())),
   Attack() );
```

To complete our set of complex predicates, let's define `pred_not`. Because not satisfying a predicate does not give us any additional information about an object, `pred_not::output_type` is the same as its `input_type`:

```
template <class Pred>
struct pred_not {
   typedef typename Pred::input_type input_type;
   typedef input_type output_type;
   Pred pred;
   pred_not( Pred p ): pred(p) {
   }
   bool operator()( const input_type* pObj ) const {
      return !pred(pObj);
   }
};
```

Besides being powerful, the complex predicates we defined are also type-safe. Consider the following example:

```
X( objects, Or(IsActor(),HasArmor()), Attack() );
```

If used with our class hierarchy, the above example will not compile. This is because if an object passes the predicate, it may or may not be of class `CActor`, and the compiler will generate a type mismatch error when trying to call `HasArmor::operator()`. However, if we used `And` instead of `Or`, there would be no compile error due to the `static_cast` in `pred_and::operator()`.

Predicate Expressions

So far, our predicate system is pretty powerful, but nested predicate expressions are not fun. We need to be able to use a more natural syntax. For example, instead of

```
X( objects, And(IsActor(),HasArmor()), Attack() );
```

we want to be able to write:

```
X( objects, IsActor() && HasArmor(), Attack() );
```

The obvious solution to this problem is to overload the operators we need for predicates. For the system to work, we need to overload the operators in a way that can be used for any predicates, including custom predicates defined further in our project.

However, if we define `operator&&` the way we earlier defined the `Or` function template, it would be too ambiguous — we need a definition that the compiler will consider for predicates only. To achieve this, we need some mechanism to distinguish between a predicate and any other type. One way of doing this is to have all our predicates inherit from this common class template:

```
template <class T>
struct expr_base {
   const T& get() const {
      return static_cast<const T&>(*this);
   }
};
```

For example, let's define `pred_and` like this:

```
template <class Pred1,class Pred2>
struct pred_and: public expr_base<pred_and<Pred1,Pred2> >{
...
};
```

With this trick, we can overload `operator&&` like so:

```
template <class Pred1,class Pred2>
pred_and<Pred1,Pred2>
operator&&( const expr_base<Pred1>& p1, const expr_base<Pred2>& p2 ) {
   return pred_and<Pred1,Pred2>(p1.get(),p2.get());
}
```

Following this pattern, we can overload operator `||` and operator `!`. Now we can build Boolean predicate expressions that follow the natural C++ syntax, while also taking advantage of the operator precedence defined by the language.

This technique of building an expression tree through operator overloads is commonly known as Expression Templates (see Veldhuizen in For More Information).

Numerical Predicates

Predicates are usually defined as Boolean functions, but we can extend our definition of predicate to include numerical predicates. This is useful for exposing non-Boolean properties of objects. For example:

```
struct Health {
   typedef CActor input_type;
   typedef input_type output_type;
   int operator()( const input_type* pObj ) const {
      return pObj->GetHealth();
   }
};
```

Of course, the predicate version of the `X` template function treats all predicates as Boolean. We can use the `Health` predicate directly, but then we would only be able to check if the health of an actor is not 0 (or we can check for 0 if we use `pred_not`). Obviously, we need to be able to check for other values as well.

So, we can define the following predicate:

```
template <class Pred,class Value>
struct pred_gt: public expr_base<pred_gt<Pred,Value> > {
   typedef typename Pred::input_type input_type;
   typedef typename Pred::output_type output_type;
   Pred pred;
   Value value;
   pred_gt( Pred p, Value v ): pred(p),value(v) {
   }
   bool operator()( const Pred::input_type* pObj ) const {
      return pred(pObj)>value;
   }
};
```

Similarly to the case of `pred_or`, `pred_and`, and `pred_not`, we can overload the `>` operator to provide access to `pred_gt`:

```
template <class Pred,class Value>
pred_gt<Pred,Value>
operator>( const expr_base<Pred>& p, Value v ) {
   return pred_gt<Pred,Value>(p.get(),v);
}
```

Now, we can do something like:

```
X( grunts, Health()>5, Attack() );
```

This will execute the `Attack` functor only on the objects with health greater than 5. As any other predicate that defines `input_type` and `output_type`, we can combine `pred_gt` in complex predicate expressions. For example:

```
X( objects, IsGrunt() && (Health()>5 || HasArmor()), Attack() );
```

Following this pattern, we can overload all other comparison operators: `<`, `>=`, `<=`, `==`, and `!=`.

Additional Set Operations

The predicate expression system we just described is the core of our scripting support, but we still need to write some additional functions to make it easy to work with sets. Table 1 shows some additional function templates that we may find useful to define.

In addition, it is convenient to define operator functions for set intersection (*,*=), set union (+,+=), and set difference (-,-=). All of these functions can be defined as templates for maximum flexibility.

TABLE 1. Some additional functions that enhance usability of sets.

`All(set,predicate)`	Returns a set that contains all the elements of the input set that satisfy the predicate
`Num(set)`	Returns the number of elements in the set
`Num(set,predicate)`	Returns the number of the elements in the set that satisfy the predicate
`Any(set)`	Returns true if the set contains at least one element, false otherwise
`Any(set,predicate)`	Returns true if the set contains at least one element that satisfies the predicate
`FirstFew(set,count)`	Returns a set that contains the first count elements of the input set
`FirstFew(set,predicate,count)`	Returns a set that contains the first count elements of the input set that satisfy the predicate
`Some(set,count)`	Returns a set that consists of count random elements from the input set
`Some(set,predicate,count)`	Returns a set that consists of count random elements from the input set that satisfy the predicate

Integration with the Game

We have a powerful system to manipulate sets of objects, but to be able to do anything with it, we need to define the interface between the script and the game. For example, it could be appropriate to define a class that is the root of all scripts:

```
class CScriptBase {
public:
   void RegisterObject( CRoot* pObject );
   void RemoveObject( CRoot* pObject );
   virtual void Tick( float deltaTime )=0;
   ...
protected:
   set<CRoot*> m_Objects;
   set<CRoot*> CheckArea( const char* areaName );
   ...
};
```

The public section of our class contains functions that the game can execute. `RegisterObject` is called from the constructor of `CRoot` whenever a game object is created. The task of `RegisterObject` is to filter out any objects we do not want the script to have access to, and include all other objects in the `m_Objects` set which is accessible by the script. Similarly, `RemoveObject` is called from the destructor of `CRoot` to make sure `m_Objects` does not contain pointers to invalid objects. The game also calls `Tick` on each frame to let the script do its job. We can continue along these lines, but obviously there is not much the game has to know about the script.

The protected section of our class has functions that the child script classes can use to query the game for information. All such functions are implemented by `CScriptBase`. In our example, `CheckArea` will check a named area in the level for any objects and return a set that contains them. Once the script has the set, it can use the predicate system to get the information it needs. For example, to check if the player has advanced to a given area, we can do something like this:

```
if( Any(CheckArea("Area1"),IsPlayer()) )
   SignalPlayerIsInArea1();
```

Besides query functions, it is also useful to define functions that make it easier for the script to perform common tasks. This could include, for example, the ability to register a member function of the script for automatic periodic execution.

Defining Additional Templates

To operate properly, our predicate system depends on the `select_root` and `select_child` templates. The C++ language does not provide direct support for something like that, but we can trick the compiler into doing what we need with some meta programming.

We will need to associate a numerical identifier with each class of our class hierarchy. To do this, we can define the following templates:

```
template <class T>
struct get_id {
   enum {value=0};
};
template <int ClassID>
struct get_type {
   typedef void type;
};
```

To associate identifiers with classes, we simply define explicit specializations of `get_id` and `get_type`. For example:

```
template<>
struct get_id<CGrunt> {
   enum {value=30};
};
template<>
struct get_type<30> {
   typedef CGrunt type;
};
```

Now, `get_id<CGrunt>::value` evaluates to 30, and `get_type<30>::type` evaluates to `CGrunt`.

In addition, let's define the following template:

```
template <class T>
struct tag {
   typedef char (&type)[get_id<T>::value];
};
```

For a given class `T`, this template defines a reference to a char array the size of the numerical identifier associated with `T`.

Finally, to continue our `CGrunt` example, let's declare the following function:

```
tag<CGrunt>::type caster( const CGrunt*,const CGrunt*);
```

Note that we only declare this function. We do not provide a definition.

For `select_root` to work, all of the classes in our hierarchy must be properly registered by providing explicit specialization of `get_id` and `get_type`, plus a declaration of the caster function. This is best done using a macro:

```
#define REGISTER_CLASS(CLASS,CLASSID)\
   template<> struct get_id<CLASS> { enum {value=CLASSID}; };\
   template<> struct get_type<CLASSID> { typedef CLASS type; };\
   tag<CLASS>::type caster( const CLASS*,const CLASS*);
```

Now let's register `CRoot`, `CActor`, `CGrunt`, and `CAgent` by invoking the `REGISTER_CLASS` macro, using a different numerical identifier for each class:

```
REGISTER_CLASS(CRoot,10)
REGISTER_CLASS(CActor,20)
REGISTER_CLASS(CGrunt,30)
REGISTER_CLASS(CAgent,40)
```

LISTING 4. Defining `select_child`.

```
template <class T,class U>
struct select_child
{
   typedef
      typename meta_if<
         get_id<typename
select_root<T,U>::type>::value==get_id<T>::value,    //if
         U,                                           //then
         typename meta_if<                            //else
            get_id<typename
select_root<T,U>::type>::value==get_id<U>::value,    //if
            T,                                        //then
            void                                      //else
      >::type>::type type;
};
```

With the classes properly registered, the `select_root` template can be defined like this:

```
template <class T,class U>
struct select_root {
   enum { ClassID=sizeof(caster((T*)0,(U*)0)) };
   typedef typename get_type<ClassID>::type type;
};
```

Now let's follow what happens when we use `select_root` with `CGrunt` and `CAgent` (this is all done at compile time):

```
typename select_root<CGrunt,CAgent>::type* pObj;
```

We have invoked the `REGISTER_CLASS` macro for `CRoot`, `CActor`, `CGrunt`, and `CAgent`. As a result, now we have the following function declarations:

```
tag<CRoot>::type caster( const CRoot*,const CRoot*);
tag<CActor>::type caster( const CActor*,const CActor*);
tag<CGrunt>::type caster( const CGrunt*,const CGrunt*);
tag<CAgent>::type caster( const CAgent*,const CAgent*);
```

In our `select_root` template, we define `ClassID` as the size of the type returned by `caster((T*)0,(U*)0)`. In our example, `T` is `CGrunt` and `U` is `CAgent`. Since we have not declared a version of the caster function that takes `CGrunt*` and `CAgent*`, the compiler automatically picks the best match from the ones we did declare, which is:

```
tag<CActor>::type caster( const CActor*,const CActor*);
```

This defines `ClassID` as `sizeof(tag<CActor>::type)`. If you recall how the tag template was defined, `tag<CActor>::type` is a reference to a `char` array of size `get_id<CActor>::value`. Since `get_id<CActor>::value` is 20, `select_root<CGrunt,CAgent>::ClassID` will also be 20. Now we simply use `get_type<ClassID>::type` to retrieve the class the number 20 identifies, which is `CActor`.

So, if we return back to our example,

```
typename select_root<CGrunt,CAgent>::type* pObj;
```

`pObj` will be defined as pointer to object of class `CActor`.

And finally, Listing 4 shows how we can define `select_child`. Here, `meta_if` is a template that's commonly used for meta programming. I'll skip its definition, but assume that `meta_if<CONDITION,T,U>::type` is defined as `T` if the condition is nonzero, and as `U` otherwise.

The implementation of `select_root` and `select_child` could be simplified if C++ had compile-time `typeof()` fuctionality. The emulation of `typeof()` through type registration used here was first discovered by Bill Gibbons (see For More Information).

Safety and Performance Considerations

One of the most important features of our scripting system is that it is type-safe. Thanks to the `input_type` and `output_type` each predicate defines, the compiler knows the class of the objects that pass a given predicate expression and will issue error messages if we try to use incompatible functors with

them. Predicate expressions that make no sense — for example, `IsGrunt() && IsAgent()` — will not compile. In addition, predicate expressions will benefit from the compiler's expression short-circuit logic.

To further improve safety, we can avoid using pointers in our sets. In this case we can convert back to pointers just before we call functors and predicates from our `X` function (or any other helper function that works with sets). Note that only one conversion to pointer per object occurs, regardless of how complex the predicate expression we use is.

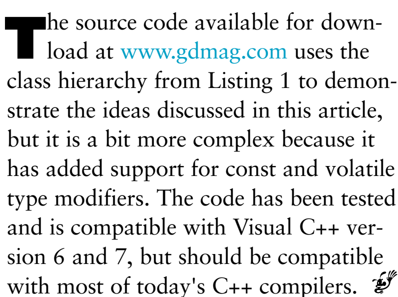
We can further separate the script and the game code by putting them in their own namespaces. Thus we can control what to hide from the script, and what to expose by providing functors and predicates.

Most of today's compilers will optimize any of our predicate expressions to inline code as if a programmer wrote a custom if statement to check for the condition of the predicate. This means that the performance of our scripting system depends mostly on the implementation of `std::set` and the iterator classes it defines.

The C++ standard mandates that the iterators of `std::set` are not invalidated when adding or removing elements from the set, which usually means that each element of the set is allocated as individual heap block. Because the elements of our sets are simply pointers, this translates to a waste of memory, increased heap fragmentation, and overall slow processing of `std::set`s due to cache misses. In addition, copying a `std::set` of pointers is a relatively slow and heap-intensive operation.

Even if we do not do anything to speed the system up, our scripts will most likely execute faster than if we used an interpreted scripting language. Still, we can speed them up significantly by using a custom allocator with the std::set class template, or by designing our own, faster set container.

Example Source Code



The source code available for download at www.gdmag.com uses the class hierarchy from Listing 1 to demonstrate the ideas discussed in this article, but it is a bit more complex because it has added support for const and volatile type modifiers. The code has been tested and is compatible with Visual C++ version 6 and 7, but should be compatible with most of today's C++ compilers.

FOR MORE INFORMATION

T. Veldhuizen. "Expression Templates." *C++ Report* Vol. 7, No. 5, (June 1995) www.osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html

B. Gibbons. "A Portable typeof Operator" www.accu-usa.org/2000-05-Main.html

ACKNOWLEDGEMENTS

I would like to thank Peter Dimov for his help in implementing and further enhancing the predicate expressions system described in this article.

2015's MEDAL OF HONOR: ALLIED ASSAULT

MEDAL OF HONOR: ALLIED ASSAULT (MOHAA) is an example of how a small team can use licensed technology to create a good game in a reasonable amount of time. Starting with a team of 11 developers, most of whom had never worked on a full title, 2015 was able to generate excitement for the game at E3 2001 and follow through with an on-time completion and both critical and commercial success. Early development on the project raised our publisher's expectations and led to a tremendous push for a strong E3 demo. Screenshot releases built up some hype for the game, and our publisher seemed impressed with the milestone builds they were circulating regularly. At E3, they turned a small viewing theater into a mock-up of a Higgins boat with netting and a "loading ramp," where MOHAA was shown to a long line of people. Even after long waits to see the demo, people were clearly enthusiastic about the game. The team was extremely gratified that we could make that kind of impact among the sensory overload of E3, and the expectations

MIKE MILLIGER | *Mike is the lead programmer at 2015.*

and excitement generated there set the bar for the rest of the project.

One of the most interesting times of the project came right after E3. Our publisher sent Captain Dale Dye to 2015 "to whip those bastards into shape." Capt. Dye is a war veteran who acted as the technical director on many Hollywood projects including *Platoon, Band of Brothers*, and *Saving Private Ryan*. Working with Capt. Dye felt like walking into the first scene of *Full Metal Jacket*.

After an afternoon of being called maggots and being forced to do push-ups for saying "gun" instead of "weapon," Capt. Dye gave us the goods: K-factors of historical weapons, realistic death animations, and tactics for our AI. He also reminded us of and reinforced our goals for realism in the design. He highlighted the unrealistic nature of many game missions where play ends as soon as the last task is accomplished, even if the player is standing in the midst of a fully alerted enemy base. Capt. Dye's guidance led us to stress the exfiltration as well as infiltration side of military operations, which made our game levels more realistic as well as more creative.

The daylong session ended in a debacle that was kindly referred to as team-building exercises. Capt. Dye wanted us to do 70 atomic crunches in two minutes. (Atomic crunches consist of everyone sitting in a row linking arms to shoulders and doing more sit-ups as a group

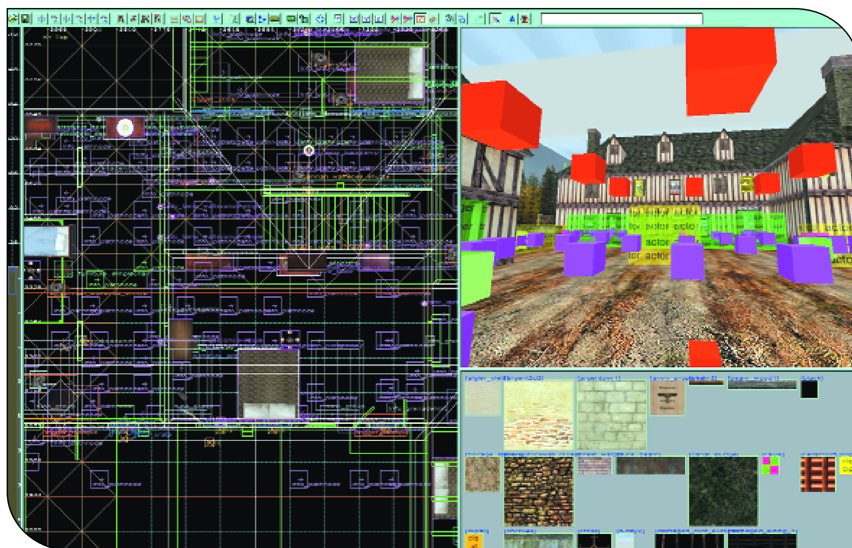
GAME DATA

PUBLISHER: Electronic Arts
NUMBER OF FULL-TIME DEVELOPERS: 27
NUMBER OF CONTRACTORS: 2
LENGTH OF DEVELOPMENT: 19 months
RELEASE DATE: January 22, 2002
DEVELOPMENT HARDWARE USED: 800MHz Pentium IIIs with GeForce 2 GTS cards
DEVELOPMENT SOFTWARE USED: Visual Studio, 3DS Max, Photoshop, in-house level editor
NOTABLE TECHNOLOGIES: QUAKE III engine, Ritual level editing tools, LIPSinc technology
PROJECT SIZE: 469,646 lines of code, 659 files

than you could by yourself). I didn't have the heart to tell him that most of the guys here at 2015 hadn't done 70 sit-ups in two years, much less two minutes.

The hours the team had to put in for the E3 demo were not insignificant, and alpha rolled around in no time, followed even more quickly by beta. It was tough to stay focused after the amount of time we spent on the E3 demo, and the company put in crazy hours on a weekly basis.

Fortunately, all the long hours paid off — with the industry focus of the past couple of years being on multiplayer games, whether massively multiplayer or mods of popular older games of the past, it was refreshing to be part of the resurgence of the single-player experience. Our hard work on the single-player game was rewarded with feedback from some that MOHAA was the best single-player game since HALF-LIFE.



MEDAL OF HONOR: ALLIED ASSAULT's level editor in action.

What Went Right

1. Level designer support. Because of the importance 2015 places on gameplay, one of the most important things we did was support the designers of the game. We licensed technology specifically for designers and used an editor that all of our designers were familiar with. In our experience, designer technology is just as important as the game engine you choose to license. The level design tools we used gave us a higher level of interactivity than what was available in a clean QUAKE III code base. Inside the game there were editing menus available to set and modify particle systems, view and scrub animations, and manipulate sound triggers and effects.

The most important feature we extended, and something that 2015 is now known for, is the scripting system. We heavily modified the scripting language to enable the level designers unprecedented control over events in the game. There wasn't anything that a designer couldn't do in the scripting system that a programmer could do in code. A designer could micromanage complete firefights and could simulate the AI system in its entirety with our scripting system if needed. We also implemented a design technique called manvis, similar to the concept of zones in UNREAL, which allows the designers to set the visibility areas in the level manually. This technique makes our open-style levels possible — without it, the popular Omaha Beach level could not have been made.

2. Feature management. Feature management is an interesting part of the development cycle and one of the hardest. Creative individuals get attached to their creations. This is a natural tendency, one that exists in all parts of software engineering, not just games.

When we announced that the Bridge of Remagen level had been cut and the flamethrower had been axed, our fledgling fan base was crushed. Without even playing the game, some proclaimed that our product was going to suck. The risk of sucking notwithstanding, we decided to cut features that weren't going to be up to the quality we demanded for ourselves. If we couldn't spend an adequate amount of man-hours to make the best flamethrower ever, then we would cut it and focus on something else. If vehicles appear in only one tenth of your game, then don't allocate more than that share of a designer's or programmer's time to make a robust vehicle dynamics system.

Two mediocre features will never equal one good feature. We were able to estimate our payoff early enough to adjust our design and schedule to make these decisions effective. We learned to pick our battles and play to the strengths of the game.

3. Licensing technology. Licensing the QUAKE III engine was another key to our success. Tom Kudirka, our president, hired people that were familiar with the QUAKE technology. We were much more comfortable using

OpenGL as the rendering API, and it just made sense to use this engine given our experience with expansion packs for QUAKE-based games. It gave us a stable build from which to work and a couple of point releases' worth of fixes by the time we started.

Licensing our technology allowed us to begin work on more complex systems right away. For example, we started work on a skeletal animation system for our characters, in addition to morph targets for facial animation. We also implemented a terrain system into the indoor-oriented QUAKE III engine using an edge collapse strategy modeled after the ROAM terrain algorithm. The unsung hero may have been the Ritual toolset — not as glamorous as licensing the QUAKE III engine perhaps, but just as effective.

We are now convinced that it's essential for teams to know when to take advantage of existing technology. Many people believe the prime skills for game programmers are beginning to shift to evaluating new technology and choosing facets of existing technology rather than the ability to write systems from scratch. I think it's kind of silly to write whole engines from scratch when you can save the man-hours and start cranking out content for the game immediately. Most new engines built from scratch seldom look as good as the mature licensable ones anyway.

4. Publisher support. The efforts of the small, original core team won over our publisher with a series of solid deliverables climaxing with the E3 demo — by that point, they felt they had a hit on their hands. Afterwards, our publisher moved the alpha, beta, and gold dates forward to solidify the game, but we never missed a due date, and we always made an effort to deliver high-quality milestones.

Our publisher also did an incredible job with the sound. Most reviews gave the highest marks to the award-winning sound team that worked on the project.

We were also fortunate to get an engineer and a designer sent from our publisher to help during the last couple of months. The designer did most of the animation scripts for the AI, which enhanced the gameplay. The engineer helped us track down bugs and nail down default engine settings based on system specifications.

Most importantly, our publisher provided quick feedback and approval of milestones. We found this to be very important to the quality of the title. I have heard horror stories where developers don't hear from their publisher for months, only to then get a long list of changes at the last minute. This type of interaction hurts the quality of the game, which reflects on all the parties involved.

5. A competing title. Many people in the game industry view competition as a bad thing. We took the opposite view, and it fueled our drive for making the best game we could. Going up against RETURN TO CASTLE WOLFENSTEIN gave us something of an underdog attitude. We were up against the legacy of the grandfather of first-person shooters. The developers working on RETURN TO CASTLE WOLFENSTEIN had a one-year head start, the executive production experience of id Software, and an experienced group to handle multiplayer. We were going against the champion, and we were the challenger.

On the other hand, with the recent success of the *Band of Brothers* book and miniseries, the World War II

realm was a gold mine of ideas, scenarios, and resources, enough to support two very similar games with different themes (similar in the sense of the time period and underlying technology). All in all, we felt that the scope of World War II could support both titles and that the games were different enough to target different audiences. World War II and FPS fans would probably buy both. Nonetheless, we still had the underdog mentality that made us work harder. Every time there were new movies or screenshots of WOLFENSTEIN, we set out to make our content look better and our gameplay run smoother. In the end, our assumptions proved correct: both games did extremely well.

What Went Wrong

1. Art pipeline and tools. Overall, there was a lack of art asset management and accounting on the project. We used an off-the-shelf product for the code, but there really was no good method of source management for artists. Not surprisingly, this caused some problems. We did look at some other source control tools geared specifically for content, but we found them to be overpriced.

Additionally, there was poor tools maintenance throughout the project. We did not have a dedicated tools person that would build the tools continuously to make sure they still compiled and the formats didn't crash the game. We changed the internal formats of our animations mid-project, and when we needed to reexport our animation again, the tools would not compile, so we had to spend unscheduled time going and fixing the tools.

We also had a data architecture that was carried over from our licensed technology. The specific functionality should have been removed and replaced with a better system that allowed the content creators to spend their time making content instead of editing text files. Over half our animators' time was spent editing level text files that loaded specific animations for that level. The system that we were using is comparable to a programmer having to make all the animations for a character before the programmer begins working on the animation system. It just doesn't make sense for a well-balanced team.

Our artists had no shader previewer. They edited a text file and then fired up the engine to see if a shader was what the artist had envisioned. A previewer would have saved many hours going back into the engine just to see the results. A smart GUI on top of the previewer would have allowed the artists to make the shaders visually instead of using an arcane, codelike shader language. We plan to address this problem immediately on 2015's next project. Seeing our artists produce good work despite all they had to endure gave me even more respect for them.

2. Assertions and profiling late in project. Our experience underlined the fact that a robust assertion and profiling scheme should be implemented as soon as possible. If you're lucky enough to get some time for preproduction, one of the first systems to be written, besides a memory manager, is the functionality to handle asserts, list the call stack, file name, and line number. Eventually, we modeled our assertion system after a popular system for tracking bugs.

Profiling came late as well, in a classic case of putting the cart before the horse. We had completed all of the base systems, then implemented the profiler. This caused confusion as to where the actual slowdowns were occurring. With a profiler in place, any change in system functionality can be immediately tracked and bad design is more readily spotted. There are tons of references and several well-known, cheap, off-the-shelf packages for profiling. We wrote our own to cut down on timing overhead but it was unnecessary to do so.

An example of the costs imposed by our lack of a profiling system would be the continuous LOD system we added to our models, which turned out to be overkill. In retrospect, we should have gone with a discrete system and cut out the edge collapse calculations. Naturally, this would have spared a large amount of artist rework on poorly deforming models. Our engine was CPU bound, and we could have used the extra processing for other systems. A profiling system earlier in the project would have

Allied soldiers sweeping the area.

given us indication and foresight to avoid these problems.

3. Too aggressive a schedule. When I first made the engineering schedule, our initial goal was Christmas 2001. For better or worse, we planned to go head-to-head with our competition. However, an initial lack of end-game focus in task implementation wasted a great deal of our development hours. We didn't think through the AI and vehicle systems early in the project, and gameplay was late to mature, which reduced level designer productivity.

While it turned into a death march, we are proud of the accomplishments of the game. There are some guys that lived at the office, and if it weren't for that kind of dedication the game would have never seen the light of day. While culling out content and functionality to make our gold date was, as I mentioned previously, ultimately a positive thing, it meant that a lot of content and time was wasted. We had planned to implement stencil shadows, volumetric smoke systems, and pixel and vertex shaders but never did. And with a short schedule like ours, the man-hours that we took away from the game to focus on the D-Day demo really showed.

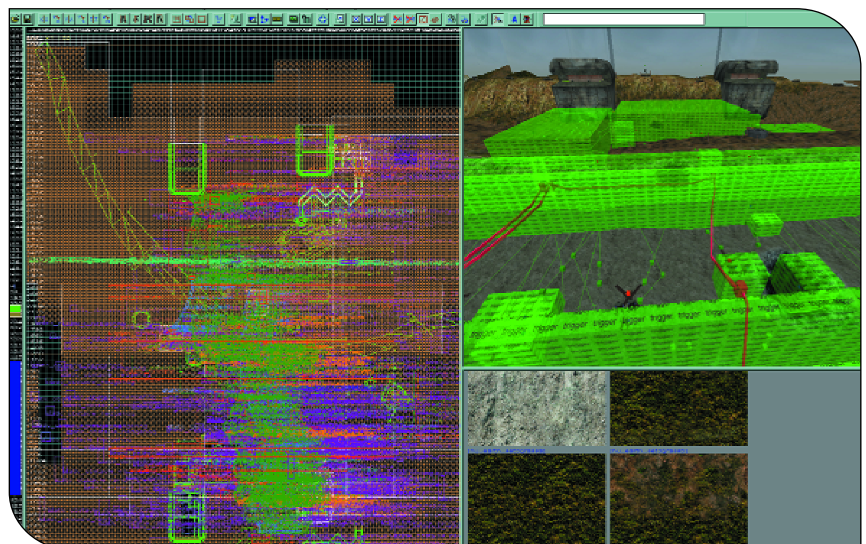
Even with these problems, we never missed a milestone and always delivered quality higher than what was expected of us. We had the game prepared a month before shipping and enjoyed a well-deserved Christmas vacation.

4. Project leadership. Because the team was small and relatively inexperienced, our structure didn't include departmental leads at the beginning of the project. We didn't have a mature process for development (especially at the beginning), and there was a lack of engineering direction (we didn't even have passwords for our source control). At the time I wrote the technical design document and made the schedule, we had three engineers. Luckily, our only animator had some programming experience, so we had to rely on him to write the foundation of our animation system. There was simply an overall lack of cohesive vision on the engineering side, so it was nice to finally get a lead in the latter stages of the project. We needed someone to pass out the deluge of bugs from the testers, communicate with the publisher about the features that were doable, and spend a lot of time talking with the producer about features that needed to be cut.

Design focus was another major problem early in development. While our technology enabled us to work with many types of gameplay, it made for an unbalanced experience. Levels in the game wavered in intensity and really did not build up to a final climax. There was no one on the project to hold the vision for the entire game, arrange the experience to steer the intensity, and assign the work accordingly. This led to a somewhat fragmented feeling for MOHAA. There was just no overall consistency in the game.

5. No evolution of graphics and burnout. Since I get to write this article, I took the liberty of reserving this last point for myself. There was no evolution in our graphics engine, which was a direct result of not having engineering vision.

The QUAKE III rendering architecture is very forward-thinking. When QUAKE III shipped in late 1999, it targeted a certain range of chipsets. But with hardware TnL coming to the consumer market after QUAKE III's release and our game shipping in 18 months, I thought that evolving the engine to support a higher range of graphics cards would be the proper thing to do, but I felt like I was alone in my desires. I had to put my PC duties on hold to do a prototype



Level designer's view of of the assault on Normandy Beach.

Historical references made conceptualizing the look an easier task.

for a next-generation console that we eventually put on hold to work on the E3 demo. That time could have been spent supporting vertex and pixel shaders, register combiners, and hardware TnL. That functionality would have taken the game to the next level.

The last 10 percent of the project is always the hardest. It's easy to begin a project and give it your all when everything is brand-new. After the crunch of E3 and the console project getting put on hold, I was burned out. I could have put in the extra time (beyond the crunch we were already working) and rewritten the renderer, but at this point in the project, I just didn't have the steam. On this occasion, I felt like I let my teammates down. I'm the type of person who learns more in defeat than victory, and I now realize that making a game is not a sprint, it's a marathon.

The Road to Experience

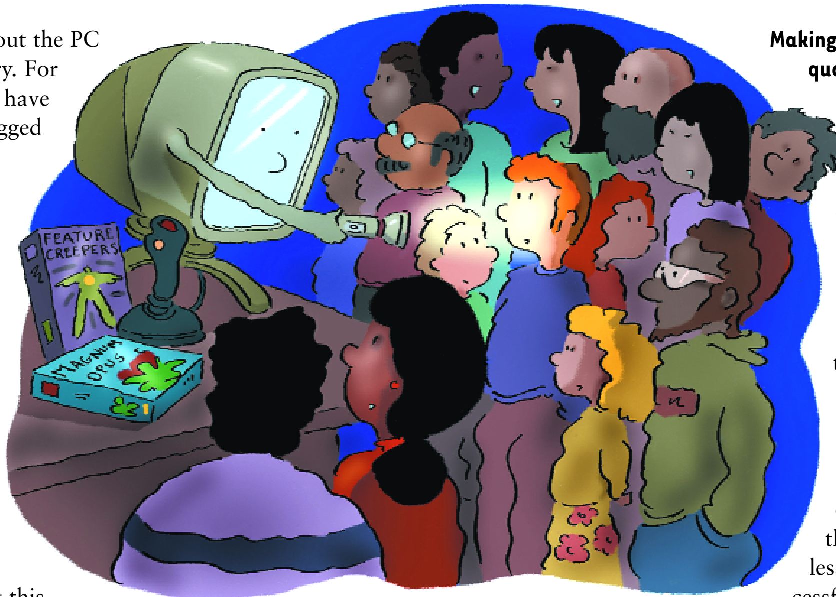
I am not a guy who pretends to have been around the game industry for years. But with a lack of experience comes a lack of jadedness, which has been to my advantage in some respects. For one, it's given me a fresh perspective on the process of creating MEDAL OF HONOR: ALLIED ASSAULT.

Even though 2015 has only been around for four years, with MOHAA we managed to identify what is fun and what people like. One of my favorite books, *Design Patterns* (Addison-Wesley, 1995), explains the difference between an expert designer and an inexperienced one: "One thing expert designers know not to do is solve every problem from the first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts." Our company has completely bought into this philosophy for making games, and we will continue to use our system and apply our formulas in hopes of achieving more best-selling titles in the future.

More Professionalism Wouldn't Hurt

(Or the Seven Deadly Sins of Computer Game Development)

I'm worried about the PC gaming industry. For years our sales have significantly lagged behind the increases in home computer penetration. Our audience, while bigger than it was 10 years ago, seems like it is made up of a fairly uniform demographic. With very few exceptions, we have not successfully reached beyond our core gaming audience to create new computer gaming consumers.



Many of you reading this are thinking, well, aren't games as big as Hollywood now? What is this fool talking about? Well, our entire industry might be as big as the North American box office sales, but we aren't anywhere near as successful as Hollywood in total revenues. PC games reach only a fraction of their audience, and worse yet a tiny fraction of the potential computer gaming audience.

In many ways we have sown a crop of marginalization that we now reap by a series of behaviors that we could change. I have assigned one of the seven deadly sins to each these behaviors. Before you get too outraged, I have to admit that I have been guilty at one time or another of all of these sins myself.

Allowing low-quality games to be released on the unsuspecting public (Greed). This is the most deadly of all our sins as game developers. Team members and product management need to be much more vocal and active in this area.

Our names are going to be on products we are not proud of if we don't stand up for the consumer before a product that's not ready for prime time goes out the door. Shipping for financial targets at the expense of damaging the franchise value and burning first-time consumers (ensuring they never buy another computer game again) is self-defeating.

Making games that only run on the latest, greatest hardware (Envy). This ongoing trend completely limits us to the most dedicated gaming consumers, and actually discourages consumers from getting involved in computer gaming. Far too many consumers have been turned off by purchasing a game only to find out that their machine can't run it. Or their machine can run the game but the experience is so poor that they think the game or their machine is broken. Designing and shipping products that require the latest drivers and hardware is a sure-fire way to ensure that games remain a hobby instead of a mass medium.

Making games that substitute quantity of features for gameplay (Sloth). Too many games use feature quantity as a design theme. The thought process seems to be that if we make a big enough game, more people will find something interesting to play within it. This is a lazy way to design games. Products that really nail their core design elements and make them clean and seamless seem to be more successful overall.

Making games too big, too complex, and too long (Gluttony). One way we try to beat the competition is by equating bigger with better. If some complexity is good, then more is better. If 40 hours of gameplay is good, then 100 hours must be better. If a large game world is good, then a bigger one must be better. While I like a bigger, longer, and more complex game as much as the next person, these are not innovative or creative design goals. People who have not played dozens of games are not necessarily impressed with bigger, longer, and more complex games, as they don't work from the same the reference that the hardcore audience does. In fact, a game that has 100 hours of gameplay may intimidate many consumers who can't imagine spending that kind of time on their entertainment.

Making games only we want to play. (Lust). While it is critical to have passion for what we do, treating ourselves as the

continued on page 71

Illustration by Scot Ritchie

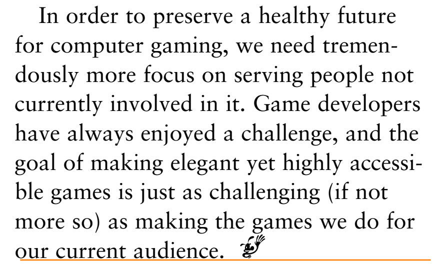
continued from page 72

core audience ensures we will only appeal to hardcore gamers. Too often we believe that if we lust after a game, everyone else will too. As professionals and (gasp!) artists, we should be expanding the reach and impact of our medium. While there is a place for elitist art within a medium, typically it is the minority of the output, since it caters to a very limited audience. Strangely enough, our industry has most of its output geared to the gaming elite, not the broader potential audience.

Spreading the blame (Anger). Too many of us blame problems with our games on the elusive "them," which pretty much includes everyone except ourselves. Favorite targets include the publisher, management, marketing, and other team members. I bet most of this anger is really anger at ourselves for not doing everything in our power to make better product before it ships.

Not letting players under the hood of our games (Pride). Too often developers don't let players play with their games or game worlds. We want them to play the game the way we would. But historically those games that let players add their creativity by adding features or data to the game extend their shelf life tremendously.

In order to preserve a healthy future for computer gaming, we need tremendously more focus on serving people not currently involved in it. Game developers have always enjoyed a challenge, and the goal of making elegant yet highly accessible games is just as challenging (if not more so) as making the games we do for our current audience.



GORDON WALTON | *Gordon has developed over two dozen games and has managed the development of hundreds of others. He is currently VP and executive producer of THE SIMS ONLINE at Maxis.*