gd

GAME DEVELOPER MAGAZINE

NOVEMBER 2002

UBM Tech

GAME PLAN

LETTER FROM THE EDITOR

Hello World

The game development world has long been dominated by the triumvirate of superpowers encompassing North America, Japan, and Western Europe, where the vast majority of development work has taken place over the last 25 years. Now, as the game industry continues to grow and expand, the gravy train is rumbling into territories heretofore not associated with professional game development, places such as Russia, the Czech Republic, Vietnam, and South Korea to name just a few.

Development in these emerging markets is much cheaper, which is a primary attraction both for publishers and for developers looking to subcontract work. Cost savings is a huge consideration as teams grow, development times lengthen, and more experienced developers and studios command bigger cuts.

Some think the corollary to this trend is "you get what you pay for," assuming that game development done in far-flung places is going to be fraught with logistical problems such as language and cultural barriers, and of inferior quality to work done by more established professionals occupying the traditional game development landscape. But with technology, physical barriers aren't what they used to be, and there are tools on the market now such as Renderware Studio and NXN Alienbrain that aim in part to keep far-flung teams better connected.

As for the quality issue, OPERATION FLASHPOINT, developed by Czech Republic–based Bohemia Interactive Studio, is a great example of not getting what you pay for in the best way. The game racked up $400,000 in development costs over four years, the kind of change that an increasing number of today's AAA games lose under their sofa cushions, and proceeded to sell 1 million copies worldwide for publisher Codemasters while garnering plenty of critical acclaim. Success in the PC market then paves the way for console work and the potential for a whole new audience; FLASHPOINT is on its way to Xbox, as is Croteam's SERIOUS SAM, which originated in Croatia. Most recently, the Czech Republic's Illusion Softworks, makers of the HIDDEN & DANGEROUS series, released the much-hyped MAFIA; it would not be surprising to see both properties end up on consoles as well.

Tangentially, MAFIA raises one cautionary point for publishers and hardware makers looking to extend their development frontiers — not the game itself but the problem in many emerging economies of organized crime. This is a real issue for companies looking to do business in Central and Eastern Europe and in some areas of Eastern Asia. Until the regulatory efforts of the governments involved stem the rampant proliferation of organized crime, due diligence can't be shortchanged when seeking out business partnerships of any kind. However, many expect at least the Central European countries to be admitted to the European Union within the next few years, which will greatly ease commerce with fair trade standards to which member countries must adhere.

Games are indeed going global, both in content and in business. Having come out in the world as far as the business has, there's no reason to think it will retreat back to the safety of traditional strongholds rather than continue to blossom in emerging markets. Any backlash or resistance from studios in more established regions would be both unfortunate and short-sighted. Recently I attended a panel at GDC Europe called "Developing in a Developing Country," and when the question arose of whether cut-rate development costs fostered profiteering among the big game publishers to the detriment of established (and more expensive) studios, Bohemia Interactive's Marek Spanel answered succinctly, "Our goal is not to be cheaper, our goal is to be better." That goal is exactly what drove many of today's best studios to their current success levels, and who did they let stand in their way?

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/Associate Publisher
Michele Sweeney e: *msweeney@cmp.com* t: 415.947.6217

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

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

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

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

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

Director of Marketing Greg Kerwin
Senior MarCom Manager Jennifer McLean
Marketing Coordinator Scott Lyon

CIRCULATION

BPA INTERNATIONAL
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
Chief Operating Officer Steve Weitzner
Chief Information Officer Mike Mikos
President, Technology Solutions Group Robert Faletra
President, Business Technology Group Adam K. Marder
President, Healthcare Group Vicki Masseria
President, Electronics Group Jeff Patterson
President, Specialized Technologies Group Regina Starr Ridley
Senior Vice President, Global Sales & Marketing Bill Howard
Senior Vice President, HR & Communications Leah Landro
Vice President & General Counsel Sandra Grayson
Vice President, Creative Technologies Philip Chapnick

CMP United Business Media

GamaNetwork

INDUSTRY WATCH

THE BUZZ ABOUT THE GAME BIZ | daniel huebner

Greek gaming ban. An effort to restrict illegal gambling in Greece took an unexpected turn when the Greek government admitted that it wasn't able to distinguish between illegal gaming machines and innocuous videogames. The resulting law decreed a ban on the use of all electronic, electromechanical, and electronic game devices in public places. The law went into effect in July, leaving players of arcade, PC, handheld, console, and mobile games in Greece to face fines of €5,000 to 75,000 and jail terms up to 12 months.

The confusing law failed its first judicial challenge in September when, in a case brought against three Internet café owners arrested in August after police found customers playing online chess in their establishments, The Court of First Instance in Thessaloniki ruled that the law contravened constitutional provisions on the free movement of ideas.

Despite publishing the critically acclaimed HEROES OF MIGHT AND MAGIC IV earlier this year, 3DO is fighting to stay listed on Nasdaq.

Hanging on to Nasdaq. 3DO and Interplay both made moves to secure their places on the Nasdaq stock exchange, at least temporarily. 3DO obtained shareholder approval to enact an eight-for-one reverse stock split intended to push the company's share price, hovering at the time around 20 cents per share, back above Nasdaq's $1 minimum share price requirement.

Interplay's position is a bit more precarious as it fights to keep its place on Nasdaq's Small Cap market. The company missed its August 13 deadline for compliance with the $1 minimum share price requirement and also failed to meet requirements of $5 million in shareholder equity, $50 million market value of listed securities, or $750,000 net income on continuous operations to become eligible for a 180-day extension. Interplay was to face de-listing on August 23, but was granted a short stay of execution when the Nasdaq Listing Qualifications Panel granted the company's request for a hearing.

STAR FOX ADVENTURES: Rare's last outing for Nintendo?

Rumors around Rare sale. Microsoft has, since the inception of the Xbox, been rumored to be involved in the purchase of nearly every developer in the industry. Word of Microsoft's interest in Nintendo stalwart Rare, however, took an unexpected turn in September when Nintendo first voiced its approval of the idea. Nintendo and Rare have had a long-standing relationship, and Nintendo holds a 49 percent interest in the studio, but Nintendo recently declined to purchase Rare's remaining shares and was know to be reconsidering the developer's Nintendo exclusivity. Citing Rare's diminishing importance as a Nintendo developer, Nintendo was at press time welcoming buyers interested in its share of the company, including Microsoft. Rare titles accounted for 9.5 percent of Nintendo's unit sales in 2001 but slipped to just 1.5 percent this year.

Console price cuts in Europe. Basking in the glow of better-than-anticipated sales in Europe, Sony announced a new round of price cuts for the Playstation 2. The price of a PS2 in Europe dropped by 15 percent in the U.K. to £169.99, and to €249 to 259 on the continent. Microsoft announced an even more aggressive price cut less than one hour later. The Xbox price was slashed from €299 to 249 on the continent, with the U.K. price hitting £159, making a new Xbox less expensive than a PS2 in the U.K. for the first time.

Sony drops Playstation 3 hints. While Sony is remaining tight-lipped about its plans for a Playstation 2 successor, signs are pointing to the company taking its console in a new direction sometime in 2005. The 2005 date comes from the estimated completion date of a four-year chip researcher project, codenamed "Cell," currently underway as a joint project of Sony Computer Entertainment, Toshiba, and IBM. Whenever the PS3 arrives, it's likely to take a different form, possibly arriving as an integrated part of other devices. "We're not thinking about hardware," explained SCE's Kenichi Fukunaga. "The ideal solution would be having an operating system installed in various home appliances that could run game programs."

UPCOMING EVENTS CALENDAR

THE AUSTRALIAN GAME DEVELOPERS CONFERENCE

MELBOURNE CONVENTION CENTRE
Melbourne, Australia
December 6–8, 2002
Cost: variable
www.agdc.com.au

DV EXPO

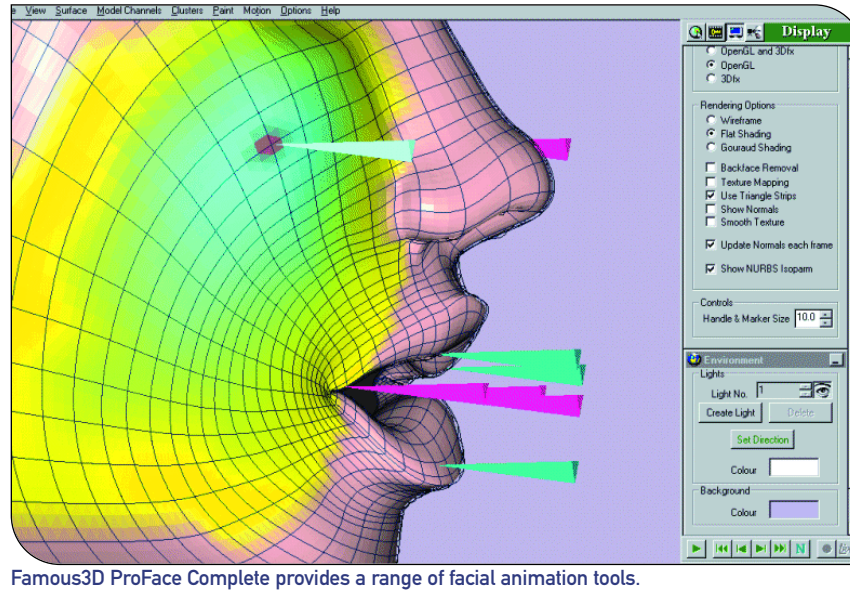
LOS ANGELES CONVENTION CENTER
Los Angeles, Calif.
December 9–12, 2002
Cost: variable
www.dvexpo.com

Famous3D ProFace Complete

by steve boelhouwer

At its heart, ProFace Complete, the latest incarnation of Famous3D Animator, is a cluster animation tool. The program has a wealth of features specifically designed for the efficient tagging, weighting, and animating of vertex or NURBS point clusters. ProFace has strong ties to a motion capture workflow, but it's also quite easy to rig and animate a head without the benefit of mocap data.

A typical workflow involves building a character using a modeling package, importing the model into ProFace, defining clusters, applying and tweaking motion data, then exporting back to the modeling package or creating a new file. Currently, ProFace Complete interfaces (via a plug-in) with 3DS Max, Maya, Softimage 3D, Lightwave, and Filmbox. Without a plug-in, geometry import is limited to .OBJ and .3DS files (with .OBJ export only). Motion data can be brought in from most popular systems, including real-time capture setups. Audio and video files can also be imported, and a text-to-speech and audio-to-speech engine allows for facial poses and lip-synch to be created based on spoken or written dialogue.



Famous3D ProFace Complete provides a range of facial animation tools.

ProFace in Action. ProFace's interface is clean and uncluttered, with a Max-ish command panel that toggles between modes. The program has no built-in help system other than a table of mouse and keyboard shortcuts, although a decent set of printed and PDF manuals and tutorials is supplied with the software.

After a head model is loaded in ProFace, the real work of building clusters can begin. ProFace uses a brush-based system, similar to Maya's Paint Effects, for this task. A brush's attributes are easily defined, although custom brush settings cannot be saved for later use. Also missing is a symmetrical or mirror painting mode, which tags mirror sets of vertices on either side of a model (such as the cheeks or jowls). Clusters are displayed as color gradients that indicate their hotspot (center) and area of influence, and are defined as one of five types: translation, rotation, double rotation, eye (for, you guessed it, eyes), or spline.

The spline cluster is something special and represents a jackpot for lip-synch artists. In this type, vertices follow the shape of a spline curve, as opposed to a single hotspot. This approach works great for animating the lip and mouth areas on a character.

Once defined, clusters are married to channels of animation data. Given its mocap heritage, strong tools exist for defining, scaling, and applying imported motion channels. Mocap data can be either 3D or 2D, as the program works with the optional vTracker module to translate video of an actor wearing facial markers into 3D animation data. A good expression and constraint system allows users to re-create 3D movement based on 2D mocap (for example, an expression

STEVE BOELHOUWER | *Steve is the vice president of creative services for The Vendare Group, a Los Angeles–based network of game, entertainment, and marketing companies. Contact Steve at steve@vendaregroup.com.*

could translate the lips forward whenever they purse together in a pucker).

ProFace also has a keyframe editor that allows for direct access to individual animation channels. Essential for tweaking existing animation as well as for creating it from scratch, the editor can also be used to define morph targets that can be utilized internally or to export deformed geometry.

When the animation is complete, the plug-ins take your model and its newly married animation channels back to the original program. Animation data can also be exported separately as a MEL (Maya Embedded Language) script or a well-formatted text file for game engine use. Web authors will be happy to learn the program can export animations to either Shockwave 3D or its own proprietary format. ProFace can also render the viewport animation as an AVI movie.

Bottom Line. ProFace's narrow focus is both its greatest strength and its biggest liability. For projects that involve a lot of detailed facial animation and lip-synch, particularly if the animation will originate from a mocap source, it's the ideal tool. But it's an expensive package, and given its lack of suitability for other purposes, it may be a tough sell for smaller studios.

To address this issue, Famous3D now offers different subsets of ProFace at varying price points and with different motion input capabilities. ProFace Voice ($995) accepts text and voice input only, ProFace Video ($1,995) supports live video capture only, ProFace Mocap ($3,995) does 2D and 3D mocap input, and ProFace Complete ($5,995) has it all. For 3DS Max users who are only interested in a tool to help them paint and build clusters, there's also FaceAce ($495), which is specific to that task. Additionally, a lower-priced line of software is aimed specifically at web animation. By offering features of Famous3D ProFace à la carte, studios that need to make their characters talk are much more likely to find a tool suited to their needs.

FAMOUS 3D PROFACE COMPLETE ★★★★

STATS

FAMOUS3D
San Francisco, Calif.
(415) 835-9445
www.famous3d.com

PRICE
$ 5,995 (MSRP)

SYSTEM REQUIREMENTS
Pentium 133 or higher CPU with 128MB RAM, OpenGL accelerator, and video capture board for motion capture.

PROS

1. Powerful cluster animation tools.
2. Flexible and open mocap interface.
3. Multiple data export options.

CONS

1. Expensive.
2. No integrated help system.
3. No Softimage XSI plug-in yet available.

APPLE TITANIUM POWERBOOK G4

by david stripinis

Few can look at Apple hardware like the sleek Titanium Powerbook without admiring its physical beauty, but Apple has always been handicapped by a perception that there is no software available for it. While you definitely have more options available on Windows, Lightwave, Maya, and at last Photoshop all have OS X versions available. (3DS Max, however, remains available only under Windows.) The loss of some applications gains you access to a variety of Apple-exclusive applications, including — most notably for game developers — Final Cut Pro, Apple's exquisite video-editing software. Apple has recently released a version of Shake, a high-end compositing product for OS X, at a significant price reduction over the Windows and Linux versions.

My review unit had an 800MHz PowerPC G4 processor and 512MB of RAM, a setup that retails for $3,199. The base model Titanium Powerbook has a 667MHz G4 processor and 256MB RAM for $2,499. Both come with ATI's Radeon Mobility 7500 with 32MB of DDR video memory. Other custom-built options are available direct from Apple.

When discussing the hardware, before you even turn it on, you just want to sit back and admire the artistry of the industrial design. It measures less than one inch thick, but the titanium shell feels incredibly solid.

Looking around the edges of the machine, it appears to lack ports. Looks are deceiving, however, because under a clever flip-down rear plate the Powerbook reveals two USB ports, nine-pin Firewire, a gigabit Ethernet port, as well as a built-in modem. There is also built-in 802.11b connectivity (standard on my review model, an add-on on the lower-priced version), though I did find the reception to be spotty compared to some other computers. For anything else you may need, a PC card slot allows for future expansion.

Not just pretty on the outside, the TiBook has real juice to it. Photoshop 7 zips along with ease. It's hard to get any real work done with a touch pad, but you can easily add a mouse or tablet, both of which travel well. In fact, I know of one artist who brings his Powerbook and tablet to life-drawing sessions instead of a sketch pad.

Maya is currently available for Mac OS X in version 3.5, though Alias|Wavefront announced that version 4.5 will bring Maya into parity across all platforms. Alias|Wavefront actually recommends against using the Powerbook to run Maya, though for me everything seemed to run at a more than reasonable pace for a laptop, with only minor hassle depending on whether you are working with a one-, two-, or three-button mouse. Also, Alias|Wavefront does not offer the Unlimited version of Maya for the Mac, only the Complete package.

Apple recently released the DVI to ADC adapter, which allows Macs with DVI connections (like the Titanium Powerbook) to use their line of fantastic Studio Display monitors.

As for battery life, the TiBook had incredible legs, as long as I wasn't using the hybrid CD-RW/DVD-ROM drive constantly watching movies or playing

excellent
very good
average
disappointing
don't bother

audio CDs. Average life on a single charge was approximately 3 hours and 45 minutes. OS X includes a relatively accurate battery life utility that not only tells you how much time you have left while operating on a battery, it tells you how long till you have a full charge while running off the AC adapter.

Putting a Mac person and a PC person in a room to debate which is faster hardware is like putting two wet badgers in a sack. When it's all over, you can't really tell who won, and all it leaves is a big old mess. I'm not going to lie to you — the G4 is not as speedy as your latest Pentium or Athlon, and the ATI Mobile Radeon does not give the polygon performance of a Quadro4 2 Go, never mind a desktop workstation–class video card. But for 3D modelers and animators working with high-density geometry, this is not a workstation replacement. It's a great system to use on the road, or as a personal system.

For texture artists, sound designers, and programmers, it has all the power you need, with the style and user interface Apple is famous for. Add in the new capabilities of OS X and you have a platform ready for anybody. This is best evidenced to me by the fact that at the company I work for, our lead artist just bought one, and our technical director (of the programming variety, not the art kind) was planning to get one within a couple weeks. A computer that satisfies the needs and wants of not only an artist, but a programmer? Inconceivable.

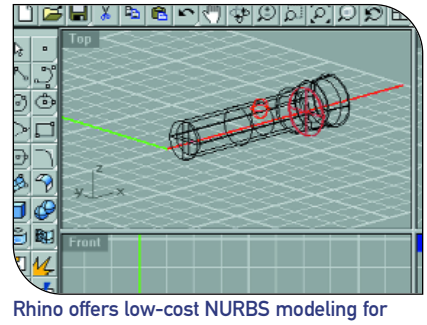
★★★★ | Titanium Powerbook G4
Apple Computer | www.apple.com

David Stripinis is director of animation at Factor 5. Contact him at david@factor5.com.

ROBERT MCNEEL AND ASSOCIATES' RHINOCEROS 2.0

by tom carroll

Using NURBS, almost any object that can be imagined can be described as a combination of relatively simple forms. Finding those forms is up to the individual artist or designer, and bringing them together is made much easier with Rhinoceros 2.0 — affectionately called Rhino by those who use it — a 3D design package developed by Robert McNeel & Associates. For the price, a relatively modest $895, Rhino is the highest-quality, most feature rich modeling package on the market today.



Rhino offers low-cost NURBS modeling for Windows.

As a pure surface-modeling package, Rhino is easy to learn and simple to use. The package's powerful import and export features that allow it to be connected to other programs such as Adobe Illustrator and IGES.

With Rhino, all curves and surfaces, including the surfaces of solids, are NURBS-based. The package supports an impressive set of tools for creating surfaces, including edge curves, planar, extrudes, lofts, networks, rail sweeps, revolves, drapes, height fields, blends, and offsets. Once a surface has been created, it can be trimmed, split, moved, rotated, scaled, filleted, chamfered, and/or copied, as well as joined to other surfaces. It's also possible to alter surfaces on an elemental level by editing their control points. Rhino has an equally impressive set of methods for creating and working with curves.

To a first-time user, however, the first thing about Rhino that makes an impression is its rather modest system requirements: A Pentium, Celeron, or higher processor, Windows 95/98/NT/Me/2000 for Intel or AMD, 40MB disk space, and 64MB RAM (more is recommended, as is an IntelliMouse).

The default user interface is divided into four views (Rhino 2.0 allows an unlimited number), and toolbars are in the places most 3D modelers expect them to be. Rhino's refresh rate is impressively fast, even with shading turned on. All of the control mechanisms you'd expect to find are present, including an array of filters, object and grid snaps, construction planes, and layers.

A host of friendly features are new in Rhino 2.0, including the ability to add plug-in applications. One such plug-in is Flamingo, an advanced raytracing renderer also from McNeel & Associates (available bundled with Rhino 2.0 for $1,195). Additional plug-in programs (most not directly applicable to the videogame field) are available from third-party sources.

Release 2.0 also includes enhancements to Rhino's built-in renderer and adds new options and capabilities to many tools for creating, analyzing, and modifying curves and surfaces. It also introduces VBScript and JScript support.

Last, but certainly not least, is The Zoo. The Zoo provides for efficient use of Rhino licenses within workgroups (Rhino 2.0 can be installed as either a workgroup node or stand-alone license).

McNeel (and his associates) should consider adding a feature tree in Rhino 3.0 that will help while editing composite geometry. Most other complaints relate to folks using Rhino for architectural purposes (such as providing comprehensive 2D drawings and cross-section views of 3D models); game developers probably won't even be looking for the package to do such things in the first place — problem solved, case closed.

In a nutshell, Rhino 2.0 provides great value for a singularly small dollop of dough. It's the perfect choice for taking your modeling to an even higher plane. Unfortunately, those who believe that anything powerful must come with a powerful price tag may just overlook Rhino.

★★★★½ | Rhinoceros 2.0
Robert McNeel and Associates |
www.rhino3d.com

Tom Carroll is lead level designer at Vision Scape Interactive. Reach him at tcarroll@vision-scape.com.

Gary Gygax
P&P, RPGs, and MMOGs

If he isn't the most influential person in the world of RPGs, Gary Gygax certainly belongs in the pantheon of the giants of the genre. He started out by writing his own pen-and-paper (P&P) games during the late 1960s and early 1970s, and went on to revolutionize gaming with his involvement in the creation of *Dungeons & Dragons*. Now, Gary Gygax is looking to bridge the gap between pen and keyboard by throwing his LEJENDARY ADVENTURES P&P RPG into the MMOG ring (currently being produced by Dreams Interactive). Ion Storm project director Harvey Smith, being an RPG fan of both electronic and paper varieties, seemed like a natural — and very willing — volunteer to talk to Gary about RPGs, MMOGs, and the transition from P&P to M&M (mouse and monitor).

Gary Gygax, legend of the RPG world, is taking his LEJENDARY ADVENTURES from P&P to MMOG.

Harvey Smith. The time seems right to launch LEJENDARY ADVENTURES as an MMOG. The commercial success of *The Lord of the Rings: Fellowship of the Ring* cannot be denied, not to mention the success of EVERQUEST. Are you happy with the game so far?

Gary Gygax. As we spent several months discussing the shape of the LEJENDARY ADVENTURES RPG online before concluding a deal with Dreams Interactive, I must say that we share the same vision, so I am indeed pleased. Of course right now there is not much game to look at. However, we have the systems and mechanics pretty well set for alpha testing, so soon the shape will be changing, as all the usual glitches and kinks in a design and graphic presentation are discovered and ironed out.

HS. RPGs often split people into several camps: Some are polarized between those players more interested in interactive storytelling and those more interested in killing monsters and collecting treasure. There are also people who play for the interesting tactical challenges, seeing the game as an extended board game. Then, of course, there are those of us who enjoy all three. How heavily will LEJENDARY ADVENTURES cater to each of the player types described above?

GG. Insightful, that question, and allow me comment on it a bit before answering.

I do not, and I stress not, believe that the RPG is "storytelling" in the way that is usually presented. If there is a story to be told, it comes from the interaction of all participants, not merely the Game Master — who should not be a storyteller, but a narrator and co-player. The players are not acting out roles designed for them by the GM, they are acting in character to create the story, and the tale is told as the game unfolds, and as directed by their actions, with random factors that even the GM can't predict altering the course of things.

Storytelling is what novelists, screenwriters, and playwrights do. It has little or no connection to the RPG, which differs in all aspects from the entertainment forms such authors create for.

LEJENDARY ADVENTURES was designed to accommodate any and all styles and play approaches, and hopefully so presented as to encourage an amalgam of all the elements of the game form. That encourages varied adventures, different challenges from time to time, and well-rounded characters (and players) that find the game has long-term interest for them. In short, I agree with you in that all aspects of the RPG should be presented and played.

HS. Role-playing games have filtered their way into the world in a number of ways. They've brought with them authorital ownership over play experiences, and also persistence to play experiences. The impact on computer gaming has been huge, and many people have been affected on a personal level as well. How do you feel about having played such an influential role in so many lives?

GG. It is a vastly stimulating thing, that impact you mention, and also quite humbling. I am always greatly heartened when I hear from fellow gamers who pass along how much enjoyment my work has brought to them, usually coupled with the camaraderie and friendships made, how much the game aided them in dealing with life and helped in attaining their potential. Had I initially realized how great the impact was to become, I would certainly have reflected on how I should present the initial work, and that might well have stifled the creativity. Still, as the positive is something well over 90 percent — more like 99.9 percent from direct communications I receive — from my current perspective I don't think I'd change a thing in regards to the concept.

HS. The sky is slate gray and purple. The clouds overhead form large faces wearing ominous expressions. Backed by this gloomy sky, a pale young sorcerer regards you warily from atop the ivy-covered remains of a shattered tower. What do you do?

GG. I waste him with my crossbow, quickly loot his body, and then complete the destruction of the tower, of course!

Toward Better Scripting Part 2

Last month ("Toward Better Scripting, Part 1," October 2002), I proposed a scripting language that automatically maintains history about the behavior of its variables, so we could more easily write scripts that ask time-oriented questions. I implemented an array of frame-rate-independent filters for each variable. The filters converged at different speeds, and I used them to estimate the mean and variance of each value over time.

Target Applications

To experiment with the history feature, I chose three time-oriented scripting tasks. I'll run through them one at a time, discussing how you can implement them with the history feature versus how you would do them without the feature.

I've included implementations of all three scripts in this month's sample code (available at www.gdmag.com). However, the scripting language is still very simple, because I wanted to keep the sample code minimal and easy to understand. The scripts use a command/argument syntax with many restrictions; you might think of them as being written in a high-level assembly language. Just keep in mind that syntactic constraints have made the scripts a little bit strange, but the semantics are much more powerful. In this article, I will use pseudocode to provide clearer illustrations.

Scenario 1: Dance Dance Revolution Commentator

In Dance Dance Revolution (DDR), you have to press buttons at appropriate times. The game gives you a health bar, which you can conceptualize as a scalar between 0 and 1. When the health bar reaches 0, you die. A commentator says things about your performance, such as "You're doing great!"

The problem with current implementations of DDR is that the game chooses the commentator's words based on the instantaneous value of your health bar. So suppose you mess up and come close to losing, but then you start doing extremely well, you're acing all the moves, and you feel really good. But then the announcer says, "You're not following the music!" because, even though you hit the last 20 targets perfectly, your health bar is still low. The commentator bludgeons your feeling of satisfaction, and he just seems broken, because you had been following the music perfectly when he said that.

Ideally, we want the commentator to detect patterns in the player's performance, rather than relying on the instantaneous value of the health variable. Not only does this increase the accuracy of comments, but it expands the scope of available comments. So he can say "Great comeback! For a while I thought you were going to lose!" which would have been impossible before.

In the sample code, script1 implements such a commentator. The game engine provides an instantaneous `player_goodness` value, ranging from 0 to 1, based on how well you hit the last target. The scripting system automatically and invisibly filters this value over time. The script queries an estimate of how well you've been doing for the past 40 or so seconds, which it uses to output a Parappa-style "You're doing bad/O.K./good/cool" message.

The script then asks for the derivative of the goodness and queries the derivative for a period of about 20 seconds. The script uses this information to choose from a range of appropriate comments, using rules such as "Was the player doing badly for a while, but is his goodness now increasing?"

The core system does not directly maintain the derivative. When you ask for the derivative, the system approximates it by taking finite differences on the history slots for the variable in question.

In one interesting case, I wanted not only the mean value of the player's goodness, but the variance as well. I wanted the announcer to say, "Great! You're unstoppable!" if you consistently perform well. One way to accomplish this result is to take many queries from the history and ensure that they are all above some threshold. But it was more natural to formulate the question in terms of "Is the long-term goodness high, and has the goodness been stable?"

All of these functions were easy to perform due to the automatic history keeping. Listing 1 shows a pseudocode fragment of the script.

JONATHAN BLOW | *Jonathan is a game technology consultant who doesn't like scene graphs. Send comments to jon@number-none.com.*

How would a programmer typically implement this scenario in the absence of a history feature? I think they would end up implementing an ad hoc averaging scheme somewhat like our history feature, but probably not so well-defined (not frame-rate independent, or not providing estimates of variance). Sometimes programmers don't think of maintaining running averages at all; they resort to storing the `player_goodness` value for each frame into a circular array and querying the array. This technique gets extremely messy and almost never works properly, since you need a lot of weird code to handle aliasing problems.

Scenario 2: Mortar Unit for an RTS Game

Consider an RTS game where a mortar vehicle fires ballistic projectiles over obstacles. Because the projectiles are slow, they are only effective when you can predict the target's position at the projectile's impact time.

Game programmers usually implement a mortar-firing heuristic in one of two ways. One way is to just fire at anything, and fire a lot. The mortar will miss much of the time, but at least there are a lot of cool-looking explosions. Recently, though, players have been clamoring for better AI; since the mortar's indiscretion in choosing its target is an obvious example of dumbness, it would be nice to avoid.

The second approach is to fire only at stationary or very slow moving targets, but this heuristic will fail in many cases. Consider a soldier in a trench alternating between two gunning positions. Overall he stays in the trench, but his instantaneous velocity is high while he is switching from one position to another. If the mortar makes its firing decision while he is moving, it will not consider him a valid target. Furthermore, a player can exploit the heuristic by, say, placing many units in a tiny but fast patrol route. The units are staying within a small area, so the mortar could easily hit them, but their instantaneous velocities are large, so the mortar doesn't consider them.

Prior to this scripting language project, I (and I think most programmers) would have tried to solve this problem with some kind of bounding volume. I'd create a circle of a fixed size centered on the initial position of each unit in the game. Whenever the unit crossed the border of the circle, I would reset the circle to the unit's current position. But if a unit didn't cross the border for some amount of time, the game considered it "not moving much" and would flag it as a mortar target.

On a full-game scale, there are a few ways to organize this computation. The way most like the real world is for each mortar vehicle to compute its own bounding volumes around all targets in the world. Unfortunately, this is $O(n^2)$ in execution time and memory usage, so we would like something faster. The most natural solution from an optimization standpoint is for each unit in the game to manage its own bounding circle. But then the implementation of each basic game entity becomes more complicated, all for the sake of one specific entity type that most of the game has nothing to do with. If you have to add this kind of uncompartmentalized handling for even a third of the entity types in your game, things quickly get messy. This also means that the game is, in a practical sense, hard to expand; if implementing a new unit typically requires changing the core entity movement code, the system is just not very modular.

A compromise between these two approaches is to implement a separate system, one not associated with entities in the game world, that serves as a "mortar brain"; there's only one of these, or one per team. The mortar brain keeps track of all the bounding circles, and the individual mortar vehicles query the mortar brain to make their firing decisions. From a software-engineering standpoint, this technique is cleaner than the uncompartmentalized version, but it's more work to implement than either of the previous approaches. If a substantial number of the unit types in your game require supplementary systems like this one, your game's complexity (and implementation time and number of bugs) will rise substantially.

All of this is an example of a common software-engineering ugliness. There's probably a name for it in some pattern book, but I will call it "overextended responsibility." The requirement for efficient computation of bounding volumes creates complexity that pollutes the entire system.

In the end, none of these bounding-volume schemes is very good. If the projectile travel time differs substantially over the mortar's range, a fixed-size circle will not be good enough to segregate possible targets. In this case, to avoid the n^2 solution, we must maintain several bounding circles of various sizes for each entity; the mess gets bigger, and the firing heuristic is only accurate to the quantum between circle sizes, so it's hard to gauge the effect of such inaccuracy on gameplay and game tuning.

By altering the firing heuristic to use position history, many problems magically go away. We fire at a target if the ellipse representing its recent position variance is very small. Already we have a solution comparable to the bounding-circle version, but since the scripting system remembers histories for us, we get everything for free. We quantify the variance by looking at the ellipse's circumference and area, as I discussed last

LISTING 1. Automatic history-keeping script.

```
recent_average = mean(player_goodness, 30);              // Mean over 30 seconds
recent_change = mean(derivative(player_goodness), 20); // Mean of derivative over 20 seconds
if (recent_average < POOR && recent_change > IMPROVEMENT) {
   output("Keep going, you're starting to get it!");
}
```

month. Because our measurements are continuous, we don't have any quantization problems like the bounding circles had. We just ask whether the variance is small enough, based on the amount of time the projectile will take to get there.

Given many targets, some of which are valid, we still need to choose one at which to fire. So we still have to deal with an $O(n^2)$ problem, but it's much simpler, since there's no bounding data to manage. A mortar brain that sorts targets by variance is so simple that it's almost not worth thinking of it as a separate system.

Interestingly, using position history solves some problems with which the bounding volumes don't help. Suppose we want to fire at targets that are moving at reasonable speeds but in predictable fashions. In other words, we want to do dead reckoning. Most games simply extrapolate the instantaneous velocity of the target. Now suppose you have a vehicle that is generally traveling west, but is constantly zigzagging in an attempt to be evasive. If you sample the instantaneous velocity, the mortar shells will land to the northwest or southwest of the vehicle, whereas anyone can see that we should target them to the vehicle's west. With position history, we easily query for the vehicle's average velocity over an extended period of time, which leads us to fire at the right spot.

The sample code script2 implements this scenario. The mouse pointer controls a target, and a mortar vehicle attempts to fire at it. The mortar shells are color coded: white shells mean that the mortar sees the target traveling in a coherent direction and is dead-reckoning it; red shells indicate that the target's movement is too erratic to predict, but it is remaining within a confined area so the shell is targeted at the center of that area. If the target is moving erratically over a large area, the mortar will not fire.

You can still confuse the mortar by moving in large but obvious patterns, such as large circles or other curves. That's because the current dead reckoning is only linear. You could implement nonlinear dead reckoning by making several queries of the mortar position and curve-fitting them. And with the finite-differencing operator I mentioned in the previous section, the implementation could become easier than that.

Scenario 3: A Tailing Mission

Sometimes you want to know if one entity has been in the proximity of another for a period of time. Maybe one entity is a proximity mine, or an unidentified cargo ship that you are supposed to scan in an X-WING–style space game. The fiction of script3 in the sample code is that you are a thief and you're supposed to follow a messenger at nighttime long enough to get a good look at him and identify him.

This is a case where we want a primitive that operates on the entire history of its inputs. We want to say `(difference_vector = the_whole_history_of(messenger_position - thief_position))`, then ask, "What is the typical length of `difference_vector` over the past 30 seconds?"

This task is easy; we can define a whole-history version of subtraction that returns a variable whose mean is the difference of the arguments' means. Some simple math tells us that this process gives the same result as if we had subtracted the two values every frame and built a history that way. So to ask our question, we just query the average of this result for 30 seconds and then take its length.

In some cases, like in the tick code that evaluates mission objectives, it would be just as easy and clean to find the difference vector between the thief and messenger every frame and put the result into a variable that builds its own history. But suppose we're not writing tick code, we're writing one-shot code that happens when an event is triggered. Say the messenger has a conversation with a third party, and the end of the conversation fires an event that asks whether you have been near enough to witness most of the conversation. With a whole-history subtraction, we perform a single query and we're done. Without history, we need to add tick code to the thief that maintains a distance measure for the event script to have a reference point. So we need to coordinate information between two scripts that run at different times, which is the kind of spaghetti code that strangles you if too much of it builds up. We ought to be able to implement a simple event without having to modify the entity tick, right?

Essentially the history-less solution is a bounding-volume implementation like Scenario 2, but it's not as nasty because the volume only concerns two specific entities and the measurement is one-dimensional. Even so, this solution suffers from discontinuities. We would have a time stamp that gets reset whenever you are standing at a distance greater than *k* from the messenger; our query would end up being something like "Is the current time, minus that time stamp, greater than the number of seconds required for success?" Success or failure is determined by a discontinuity that is hard for the player to intuit; if he steps a tiny distance over some invisible line, the timestamp resets and he probably fails, regardless of how well he did during the rest of the mission. The history solution is continuous with respect to the instantaneous distance and has some forgiveness built in (you can make up for some poor performance by exhibiting some better performance). This is probably better default behavior for most game design cases. Though there are times when you want a hard discontinuity, they're rare.

Wrap-up

I've only just touched the surface of the huge pile of time-related features a scripting language could support. I hope I've encouraged you to think about the space of features available to you when designing your next language.

Even if you never make a scripting language, the history techniques I've described can still be useful. It is not difficult to code them up in C++ or whatever your favorite language is; they won't be as transparent, but the problem-solving power is still there.

Time for a Change

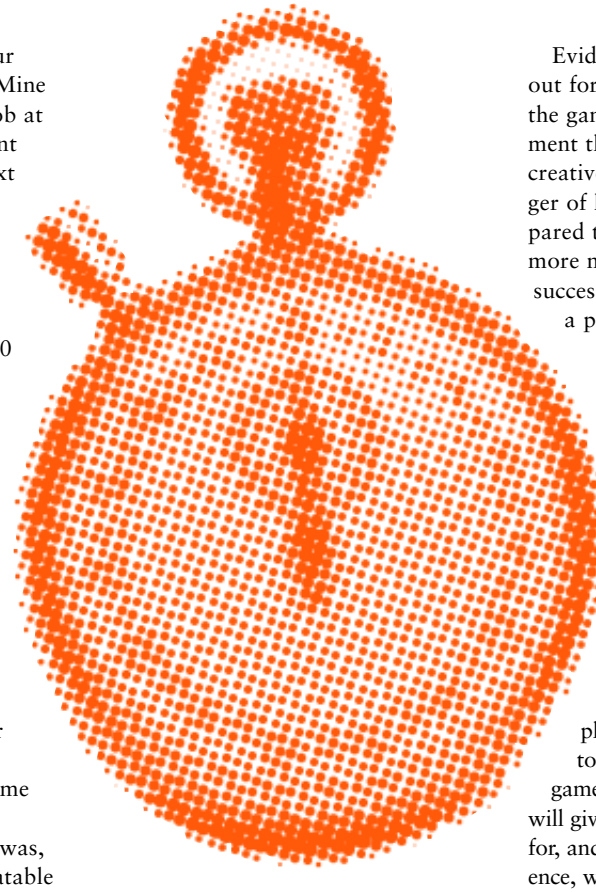
Do you remember your first job interview? Mine was for a summer job at Star Value, a discount store that lurked next to some waste ground in the less attractive end of my hometown. I was 16, and the interview was quite simple. The owner wanted to establish:

1. Was I able to move at least 100 sacks of potatoes an hour?
2. Was I prepared to act as a human garbage compactor by jumping on top of piles of discarded chicken entrails and rotten broccoli until there was more space available?
3. Would I do all of this for an hourly rate that would have appalled even my great-grandfather in 1889 as he began work as a coal miner on his 10th birthday?

Unfortunately for me, the answer to all of these questions was "yes." Where I lived, jobs were hard to come by, and at 16, I had things to buy.

As unpleasant as this experience was, it taught me (among other, less palatable things), that once I was in a position to look for a real job, I was going to have to find something that I really wanted to do (preferably something that didn't involve entrails of any description).

Fast-forward a number of years to when I was finally qualified to look for a proper job. My first choice was advertising, and applying to the big London firms required the completion of a wide range of forms and some work on mock ad campaigns. The ratio of applicants to available positions was in some cases as bad as 3,000 to 2, but by a combination of luck and ignorance, I got through to a handful of second interviews, where I realized that most advertising jobs allow you to be about as creative as a bucket of Jell-O. If that weren't bad enough, you have to use the word "synergy" quite regularly and spend at least 16 hours a day in a suit.



Evidently, this career path didn't pan out for me, and here I am working in the game industry, an area of employment that opens up a whole world of creative possibilities with very little danger of having to wear a suit. But compared to advertising, or many other more mainstream jobs, what makes a successful candidate when applying for a position at a development studio?

Getting a foot on the ladder may be the hardest step, but the journey is unlikely to end there. Any technology-driven industry is subject to pressure created by the relentless rolling forward in the world of high-tech. With the complex interweaving of hardware manufacturers, publishers, and developers, along with such elements as the console life cycle, rising development costs, and an evolving marketplace, job security is not what it used to be. Plus, few of us working in games will find that the first job we get will give us exactly what we're looking for, and as we grow in skill and experience, we naturally look for something better, or even something different.

So, as a professional game artist or animator (or both if you're showing off), what's the best way to go about finding the next job?

The answer lies partially in your circumstances and situation, but I thought

HAYDEN DUVALL | *Hayden started work in 1987, creating airbrushed artwork for the game 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.*

it would be useful to collect some information from some people actually in charge of hiring. The following is by necessity a generalization but should give some useful pointers to anyone considering a move, and is focused on the difficult process of assembling a portfolio or demo reel, which is central to any application you might make.

Preparation

Lack of preparation is where many efforts fail before they even start. You should be prepared to put in some effort before you apply. The mass-mailing approach is impersonal and easy to spot. More importantly, sending a portfolio or demo reel that has no relevance to the company or position you're looking for is a waste of time.

Presentation

As in so many aspects of life, the packaging is often nearly as important as the contents. Ultimately it is the work that counts, but prospective employers begin their assessment as soon as they hold your demo tape or CD in their hand, or the second they log on to your web site and get confused by poor or confusing presentation. Thus it's essential to recognize the difference between an effective, attractive presentation and superfluous fluff that will detract from your application rather than add to it.

Burying your work deep within a Flash-strangled site may seem like a good idea from your end ("dude, my intro is, like, eight minutes long"), but anyone who is having to sort through a stack of submissions is likely to get impatient and move on.

Content

The single most important question in putting together a portfolio or demo reel is, "What do I show?" Conventional wisdom is divided on deciding what to show, given the inherently subjective nature of any art appraisal. Nevertheless, there are a few points that are worth considering.

The showstopper. Chances are you have one Big Piece. Whether it's the hand-animated reconstruction of the barn-raising scene from *Seven Brides for Seven Brothers*, or the render you did that re-examines da Vinci's *The Last Supper* in the context of contemporary science fiction illustration, this Big Piece may not be of any relevance to the specifics of the job for which you are applying. It's designed to get the attention of the person looking at your work.

If you don't have a piece of work that fits this description, give some thought to creating one. In this case, technical limitations are of no consequence; your job is to show off. Talent is hard to quantify accurately, especially within the limited confines of a demo reel or portfolio. However, one element that can help in this assessment is a piece of unbridled creative work that showcases your eye for detail, composition, weight, balance, lighting, and so on. This piece is unlikely to get you the job on its own (and it probably shouldn't), but chances are that it will stick in the minds of those who are looking at your work.

How much stuff? In terms of animation, some people would rather see one long AVI that shows a whole variety of movement and a selection of models. Others prefer a host of concise walk cycles and attacks, each rendered out separately so they don't have to pick through extended cinematics. Where most agree, however, is on the spread of ability shown. Animating different behaviors and different creatures can cover all the necessary ground, so animations, whether long or short, need to make every frame count. In general, two to three minutes of animated work is deemed to be about right.

With modeling samples, volume is key. The more work displayed, as long as it demonstrates a range of skills (20 versions of essentially the same soldier is not advised), the more confident someone assessing the work can be of your ability.

Textures are often assessed in the context of a model, but showing original textures in their raw form can be useful. If 2D skills are a definite part of the position you are looking for, do not leave this area of your portfolio too austere.

What should you show? When asked what they'd like to see as part of a portfolio or demo reel, many managers agree on some key points. First, they expressed an emphatic "no!" to spaceships. If you've been in games for a while, you may have noticed the groundswell of anti-spaceship opinion (you might even be a leader of the movement), but whatever the case, stay away.

Why? As far as difficulty goes, spaceships are at the easier end of the spectrum, and while a genuinely good spaceship is not as easy to achieve as some might think, I have to bow to the majority in advocating a strict no-spaceship policy. Some developers go as far as to remove a demo tape at the first sign of any spacecraft. You've been warned.

The second area of consensus relates to human characters. Almost everyone I have spoken to expressed a particular interest in work that demonstrated high-quality modeling and animation of humans. Again, this relates to the difficulty involved in producing a convincing human character and animation. If you have them, you should include human models in your portfolio.

Should you include a piece of work that you have done specifically to reflect a target company's particular style or product? Some people think that this shows initiative and a genuine interest in their games; others think it's a nasty piece of brown-nosing. As a guide, if your body of work has no real bearing on the job for which you are applying, you may need to show that you are flexible enough to work in that style. If you already have this covered, don't overdo it.

Should you use high- or low-polygon models? How low is low? It's difficult to be exact about a number, but at present, "low-polygon" work seems to represent characters in the range of 1,000 to 3,000 triangles. Higher polygon counts are getting increasingly more prevalent, and work up to the 10,000-poly-

gon range is becoming the norm. One thing is vital: You should show work that you've created for a game, to illustrate that you understand the process.

Do traditional art skills have any relevance? This question provokes an almost universal "Yes." Most employers value an applicant more highly if they can demonstrate traditional drawing, painting, or animation skills, as many consider specific tools easier to learn, while a natural aptitude for traditional art is harder to come by. Therefore, if you've got it, flaunt it.

In terms of content, try to provide something different. Many of the developers I canvassed mentioned that they kept seeing the same thing. Don't undervalue originality, as long as it's relevant to your application.

Format

In terms of format, make sure your work is as accessible as possible. The people who need to see your work don't want to jump through any hoops before being able to look at it.

As far as tape goes, if you are a European working in PAL, make sure you send an NTSC tape to the U.S. (and vice versa, of course): that's just basic common sense. Drawbacks of showing demo reels on videotape include:

- They are no good for browsing static images.
- Image quality can be poor or inconsistent.
- They often have to be played in a separate room (few people have TVs and VCRs at their desks), and this small journey can see your reel put to the bottom of the pile for "later," which may in fact be "never."
- The allure of an empty tape can incline an animator to fill it with more than is prudent.

CDs are more accessible these days, and almost everyone owns a burner. A CD should be able to hold enough material for a demo reel, and if it doesn't, you might want to consider being more rigorous in your selection. Again, don't feel pressured to fill it, and make the directory structure clear.

With the advent of broadband, web sites are increasingly useful for demo reel and portfolio purposes. The interface needs to be clear, and it's important to provide any static images at a respectable resolution (don't think that those viewing your work aren't aware that substandard images can look better when small). Present your work so that it can be appreciated, or you will arouse suspicion of its quality.

Be aware of issues surrounding AVIs that use obscure codecs. A great deal of negative feedback I got from developers indicated that any AVI submitted should be in a standard format (Windows Media Player, for example) and at the very least, you should include codecs if needed.

Other Points Worth Considering

Following are a few other points that came up in the course of my polling of hiring managers.

Group work. Try to avoid sending work that is a complex team effort, where your contribution is hard to separate from the rest. If you aren't responsible for all the work, make it clear exactly what parts you worked on.

Plagiarism. Straight copies of other people's work are not viewed favorably, especially when the fact that they are copies isn't made clear from the start. Remember that the company to which you are applying will already have a varied selection of artists there who stand a good chance of having seen plagiarized work before.

Your worst piece. You should consider your portfolio or demo reel only as good as its worst piece, so you should have a rigorous vetoing procedure in place. Get as many informed, objective opinions as possible, and don't damage a sound body of work by including a single poor piece.

Jack-of-all-trades. Strike a balance between showing versatility and flexibility, and don't spread yourself too thin. If you are weak in one area, leave it alone. It's unlikely that the company to which you are applying is going to expect perfection in every category.

Technology. Game artists have to wrestle with technology as they aim to bring maximum visual impact to their game within the constraints of their hardware and software. There are some basic industry-standard tools, such as Photoshop, 3DS Max, and Maya, but chances are that changing jobs will also require that you learn to use a new piece of software, possibly something specific to your new project. Therefore, your ability to demonstrate a grasp of not only the fundamentals behind modeling or animation but also an understanding how your job fits within the development pipeline is often more useful than being an expert in a particular package.

Tests. Be prepared to do some form of on-the-spot test. More and more companies use tests now to try to reassure themselves that your work is genuinely your own. Try not to be intimidated by the pressure. The time limitations naturally restrict what's possible, but you should aim to produce something that is competent and as specified.

Think like an employer. Looking only at the work on show, would you hire yourself? If not, what area is lacking? Most of the time, you will not get a lot of mileage from applying to the same company twice in close succession, so make the first application count by holding back until you are as ready as you can be.

Be Realistic

It's impossible to please everyone, no matter how perfect your qualifications. Whatever you decide when you put your portfolio or demo reel together, it will never be all things to all people. What you should do is convey consistency, versatility, and (hopefully) talent. Once you've done that, all you have to worry about is the interview, a subject into which I'll delve more deeply next month.

You Write "Tymghte," I Say "To-mah-toe"

Grace Jones, flat-topped diva of 1980s post-punk rock, used to refute her critics, saying, "I took a singing lesson once, and they tried to tell me how to breathe; I was born breathing! I don't need to waste my time being taught!" Around the same time, Harrison Ford, in an oft-repeated myth, was filming *Star Wars: The Empire Strikes Back*. One imagines a cold Hoth morning after a few flubbed takes, when the actor, perhaps crabby and probably without coffee, said to his executive producer: "You know, you can write this s—, George, but you can't say it."

Perhaps exacerbated by the man/woman-of-many-hats history of development teams, writing for games can suffer a similar condition. Like Grace Jones, producers, game designers, and project leaders were all born breathing. Soon, they were also all talking. So how hard can it be for these talented creators of new worlds to write concise, powerful dialogue that leads to good acting and immersive gameplay? Unfortunately, it's harder than one might think.

Writing for speech, as opposed to writing for a reading audience, is a finely tuned blend of thinking-out-loud that involves placing oneself in both the character's shoes and also the actor's. Words must convey the emotional power of the scene while remaining easily spoken and free from cliché (how many more times do you need to hear the word "very" before "soul," "heart," or "existence"?). New alien worlds filled with consonant-heavy planet and species names are, while original, ideal only when matched with a phonetic translation. Planet Kjllygian will be correctly pronounced only if "Kohl-EE-ghee-ann" is noted somewhere close by.

One simple way to increase the quality of a script greatly is to actually say the words out loud that you are writing for the actors. A "table read," in which your team monopolizes a conference room, gets some food, and acts out the script together is not only fun, it will give you a more accurate idea of your characters' development and personal presence. You know that level designer who doesn't say much? Give him the biggest role; you'll quickly know if "super-secret sugar-spun task force" can have all those *s* sounds without becoming a tongue twister. While professional voice talent can make just about anything work, a non-tangled phrase allows them to focus on an accurate emotional performance. The table read is storyboarding for voice. You couldn't imagine artists spending expensive hours on levels you hadn't sketched and planned in advance. Voice (particularly because it, too, is expensive to go back and rework) should be no different.

Similarly, pay attention to whether the writing sounds too closely related to movies most people in the target market know and love. For mature and action-oriented games, throw out the scripts to Quentin Tarantino movies, *Alien* movies, *The Godfather*, and their ilk. Unless you're making direct, pop-culture in-jokes, the impact of such homage may be dulled by its familiarity among your audience. Look into sharp, pull-no-punches films such as *Sunset Boulevard*, *All About Eve,* and *The Sweet Smell of Success*. These are just a few films from a particular era (film noir and its kin) that can provide insight into how what is unsaid wields as much power as what is said. They served as the building blocks upon which many of the current films we have come to know and love were based.

Whatever the genre (excepting education), catch your audience's ear with intention rather than explication. Humans tend to want what's elusive and eschew what is plentiful. Snag your player's attention with loaded, promising brevity in the dialogue, and they'll be more willing to suspend disbelief to enter your newly created world.

Simple tips, such as avoiding alliteration, too many vowel sounds, and, conversely, too many consonant sounds (unless it's a character trait) are self-explanatory and cannot be overemphasized. If you're using an external director, take the time to go over the script with him or her in meticulous detail. They should be delighted you are willing to explain your vision and intent. Because the director's job is to get the actor to do what you want, he or she should not want to reinvent or guess about your script once in the studio, and any preparation and consideration made in advance will pay back at least tenfold in the final actors' performances.

Most game designers and producers have a wealth of talent and imagination upon which to draw. With awareness and attention, good writing can be improved, and can allow those charged with bringing your visions to life to do the finest job possible. As clients and creators, you deserve no less, and the gaming audience appreciates ever more captivating, immersive creations.

KHRIS BROWN | *Khris began casting at Universal Pictures and joined LucasArts Entertainment Company in 1990. KBA (Khris Brown & Associates) was formed in 1998 to bring professional voice production to the industry at large. Recent projects include Doublefine's* PSYCHONAUTS, *Presto Studios'* WHACKED!, *and the film* Minority Report. *Visit www.kbavoice.com.*

The Godfather Paradox

I'm still getting a lot of e-mail about my save-game challenge ("Turn-offs," July 2002), so this month I'd like to see if I can keep things interesting by providing a few more questions than answers on a new topic.

As an early employee of both LucasArts and Dreamworks Interactive, I've heard more than my share of the old Hollywood-versus–Silicon Valley debate. At the risk of straining the analogy between movies and games yet again, I'd like to look at one difference between the two industries: the success of sequels.

As a general rule, sequels to hit movies don't do as well as their predecessors, either in box-office receipts or in critical acclaim. One rare exception was Francis Ford Coppola's *The Godfather, Part II*, which won him a Best Director Oscar that had eluded him with the original film, although both won Best Picture in their respective years. Likewise, the number of movie series that have thrived for more than three films is relatively tiny, beyond a few horror series and the venerable James Bond and *Star Trek* films. The rarity of these exceptions shows how strong the trend is.

Should we just make more games about mobsters? But the game industry is full of sequels that have done better both financially and critically than their predecessors. Despite being a much younger industry than movies, we have lots of series that have gone on for eight or more releases — ULTIMA, FINAL FANTASY, WIZARDRY, and the Mario games to name just a few. Why the difference?

For one thing, games are fundamentally about interactivity and movies are fundamentally about storytelling. Or, as an old friend liked to say, "Movies are a telling, but games are a doing." I think it's easier to provide an interesting new experience in an interactive medium, because the player's participation helps keep a sequel fresh and unpredictable in a way that passive film cannot equal. The phenomenon of *The Rocky Horror Picture Show*, a movie that with audience participation became an interactive experience worthy of dozens or even hundreds of repeat viewings, suggests that this is so.

Also, it's clear that to succeed with a sequel, you must make it similar enough to its predecessor to capture the audience it had originally, but make it different enough to avoid boring or disappointing that audience with stale repetition. The very nature of our changing technology injects some of that critical freshness into games, while still enabling developers to use the same popular characters and situations. On the N64, Mario can do things he could never do on the SNES (much less the NES), and he looks much better too. The movie exceptions also seem to benefit from technology in the form of better special effects, breathing life into the *Star Trek* films or the smash hit *Terminator 2*.

An offer you can't refuse. How can we apply this theory to game design rules? Are there some basic rules — or even one overarching rule — that can sum up the ways to make a hit sequel? My instincts tell me there are probably a bunch of different rules, both about games themselves and about the process of making games, which are likely to apply to different genres or different styles of games. I'd like to hear some suggestions of yours condensed into one or two clear, declarative sentences that will be specific enough to be useful for other designers. (Please, don't send in "Make the sequels more fun and exciting than the original.")

Mailbox. My favorite letter this month comes from Greg Costikyan, who, in addition to his online computer and mobile-phone game design work, is a successful paper game designer and novelist. He responded to my suggestion that the rule about saving games may only apply to single-player games. Greg says:

"Actually, in a multiplayer game, the rule changes somewhat; instead, it becomes 'The loss of any player should not adversely affect the experiences of the other players, to the degree possible.' How you do this differs with game type; for example, with an MMOG, it might hurt a party if a key member disappears during a fight, but most of the time, it's not a big deal for one player to sign off — the world continues unabated. In a classic board or card game, you generally have an AI take over the player's position and continue, with the possibility of another live player taking over from the AI."

Greg also suggests that a more general rule would be "A player should be able to leave the game at any time without consequences." This is a good, practical rule, particularly in the online realm. Still, isn't one of our objectives as designers to make the player regret that necessity of leaving the game? I'll include more on that topic when we next revisit the save-game question.

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.*

Modular Level and Component Design

Or: How I Learned to Stop Worrying and Love Making High-Detail Worlds

So the future has finally arrived. Technologies we've only dreamed about are closer than ever, and if you're lucky enough to be on the right project, you're knee deep in it already. With polygon counts off the charts and hardware pushing more detail than you care to create, environments can be simply staggering. But with every step forward in graphics capabilities, many find themselves wondering how to utilize those advances. The question of "Wouldn't it be cool if the engine could do feature X?" is now rephrased, "Do we have the talent to utilize feature X at all?" Nowhere in the industry is that question more apparent than with environment design.

When we can draw limitless detail, it seems limitless talent is required to create those worlds. How do we quickly generate enough consistent–quality content to fill a highly detailed world? How do we make sure those worlds can be easily modified and made flexible to design whims? How do we break up the team to handle these tasks? These are daunting problems to address, and even the most stalwart industry veterans will find themselves intimidated quickly at the prospects.

As we stare down the future, it may help to take a page out of history, from tile-based platform games. Much like Super Nintendo worlds of old, it will inevitably boil down to modular components in some way. Until recently textures provided the bulk of a world's detail, but for the foreseeable future, more and more players will expect that detail to be full-blown geometry. That's a big order to fill, particularly in the competitive realm of first-person shooters, where eye-candy often reigns supreme. However, the solutions to these problems are not limited to the worlds of the running guns, they can solve dilemmas for many genres and many settings.

Using prefabricated geometry in creative ways is key to achieving the detail levels players expect to see in their game worlds. This article will look at how we at Epic arrived at the modular solution, how we implement such a system, and the benefits and limitations of a component-driven workflow.

You May Ask Yourself, How Did I Get Here?

The modular level design solution arose from the need to have great-looking, high-detail levels without having to build and texture every nook and cranny of the environment. Asking a traditional level designer to create an environment (usually from simple tools) that holds up to artist-like scrutiny is not a practical idea. Many level designers aren't deeply familiar with high-end modeling packages, and even knowing the tools doesn't necessarily translate to creating great content. The days of aligning textures and moving on have themselves moved on.

Asking traditional artists to design an engaging level is not an ideal solution either. Doing so may produce great-looking yet highly unpredictable gameplay. Questionable gameplay and performance sacrifices arise regularly when artists focus on creating large masterpieces. And both systems easily end with the creator losing inspiration after weeks of working on the same content.

Another system many developers work with is having a dedicated level designer block out an environment and pass that progress on to an artist for final polishing. This works to a degree, but it can still benefit from modular design decisions made early on. Model-

LEE "EEPERS" PERRY | *Lee is currently a designer and artist for Epic Games. Among numerous other projects, previous work includes art and level director for Ion Storm's ANACHRONOX, and modeling for Square's R&D department. He can be contacted at lee@epicgames.com.*

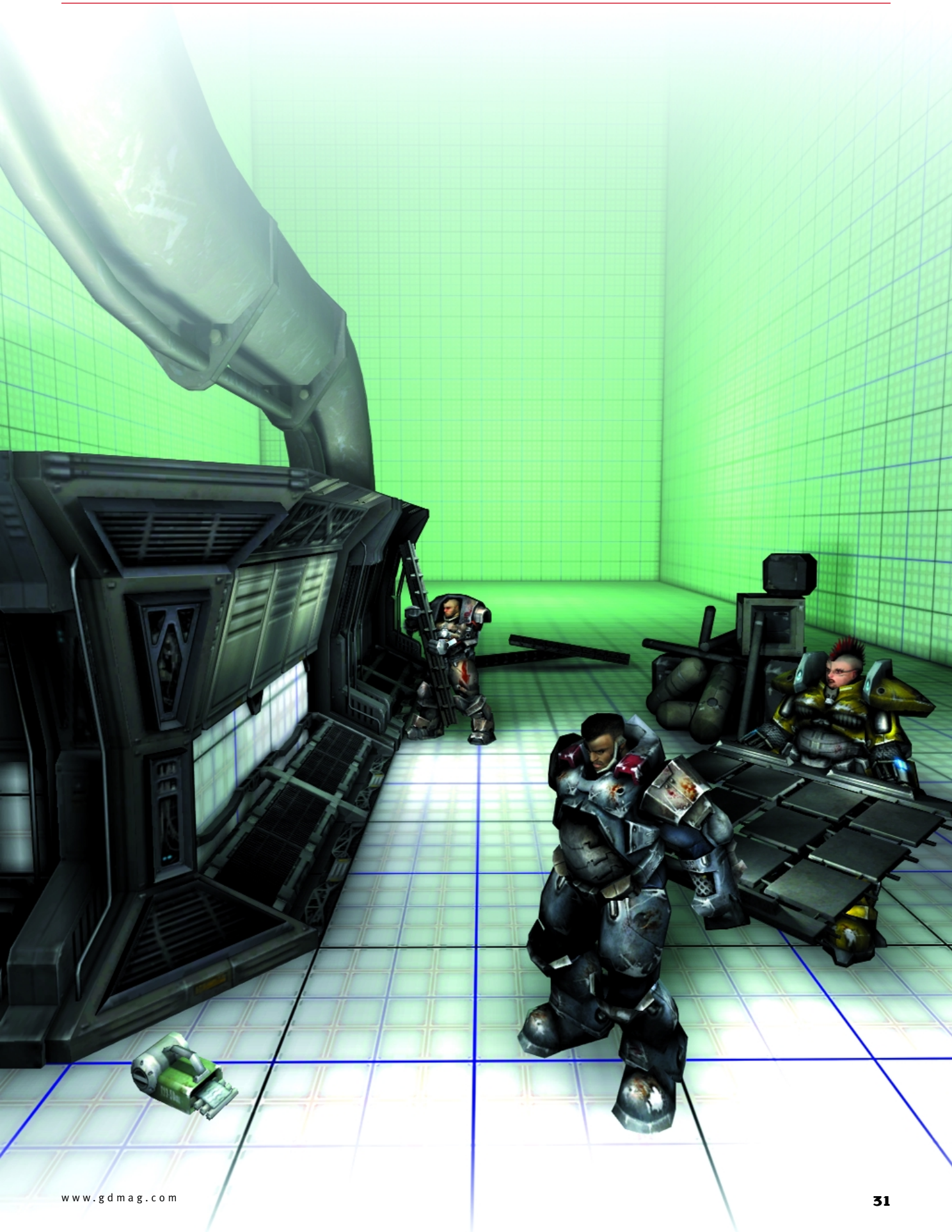


FIGURE 1A. A larger-scale building set.

FIGURE 1B. The scale has been taken down a notch, using more components in a single wall.

ing and texturing an entire level at extremely high detail levels can result in many months of work that may end up needing significant retooling. Eventually someone undergoing such a task will end up aiming to reuse as much artwork as possible from one area to the next; why not keep that tendency in mind from the beginning and work with it as a foundation for your creative process? Following are some of the considerations we keep in mind when designing our environment construction process around modularity.

Scale

With intelligent planning, modularity is not as restrictive as you might think. Indeed, modularity can be implemented on many scales. Some may choose a path along the lines of MORROWIND, wherein large buildings are single prefabricated items, and entire castles may break down into a handful of wall pieces. Other projects, with an eye on very dense environments, may work on a smaller scale that incorporates several modular components to create a single wall's detail, as shown in Figures 1a and 1b.

Where do you start when building a set of prefabs? First you must decide the scale of modularity your project is going to need. This can generally be determined by the scope of your environments. If you're going to be shoving a player through entire cities in a Porsche, you'll want to break everything down into chunks of geometry as large as entire city blocks. If you're going to be creeping a single player through the narrow, abandoned corridors of a derelict spacecraft, you'll want to work at a very fine detail level with layers of components interacting in a complex fashion.

Grids

Regardless of scale, when creating a set of modular components, nothing is as important as the almighty grid. Grids may fluctuate wildly between engines and projects, but whatever system you decide on, stick to it religiously, and always use even divisions of that grid when working on smaller components. If your system dictates that a generic wall be 256 units tall, try to keep smaller details at 128 units, or 64 units, and so on.

Be aware of game mechanics when determining the system. If a character is 128 units tall, or a leaping character can cover 64 units forward, your system should take advantage of those values. Working on a gameplay-conscious grid aids the level design process greatly, while significantly reducing "trial and error" early testing.

Keep animations in mind. Characters may interact with computer panels, use door handles, be able to sit on surfaces, or perform any number of other interactions. Deciding in advance at what height various animations will happen can save you a great deal of pain later. Of particular importance here is standard stair heights and depths. Lock down these standards early, print up guides, and distribute them as soon as possible. A few hours' work here will save weeks of potential headaches during later production.

Planning

Now's the time to sit down with the designers and artists responsible for a given area of the game and discuss what type of pieces will be required. Make a list of key features for the environment and the unique components that define the level. These could be anything from a vast missile hangar to a complex volcano mesh.

Leave the specialized pieces aside, focusing on the meat of the environments, the areas where players will spend the majority of their time. Decide which key modular components are needed and plan out the general use of those pieces. Make a shopping list of components needed for a flexible base set; this could range from hallway sections to generic city streets.

The challenges in this step often come down to transitions. Plan out how to blend one theme into another, and save effort by clarifying which transitions won't be needed. Note which structures may need capping off, and give a thought to how that should be done. For example, if you have a modular river running through your game's landscape, have a method handy for capping off that river by making it run into a grate or sinkhole, or dissipate over a waterfall. If you're building entire buildings from prefab corridors, make sure you've got a piece to cap off that structure.

Your system could have conceptual artists start their work at this point, or they may be involved earlier and have concepts presented when the initial shopping list is created.

Start with the Basics

At this critical point, be sure to go through the prefab list methodically, building the necessary components on the agreed-upon grid system. Don't begin construction with a complicated junction piece, start with the basics and create the base unit from which variations can be created. If you're making a 3D terrain set, start with the base tile, duplicate it, and create variations off the mother piece. After the various themes and sets are completed, lay them out and begin "tweening" them and creating any transitional prefabs needed. Establish an approval process at some point in the basic construction, so you don't end up with a polished set of geometry only to find out the basic theme isn't quite right. Sign off on work along the way.

Test-map your work by periodically dropping what you have created so far into a test environment and verifying that your work tiles correctly and works well together. As a bonus, you may find it inspirational to see complex geometry emerging as you work.

If your project allows it (and most do) try to keep multiple purposes in mind for your work as you go. A cave system may have ceilings that could easily be used as floors. Typical metal-plated bulkheads will often look good regardless of orientation.

Don't assume a large structure has to be a single modular piece. For example, instead of building an entire modular corridor, build modular wall sections, ceilings, and floor tiles that snap together to form a complete corridor. This will allow for more flexibility when the set gets into the designer's hands.

It also helps to provide plenty of variations. If you have a large stairway piece, make short, medium, and large versions for added flexibility, and be sure to consider texture variations as well. With a change of color scheme, an office hallway may appear completely different. By incorporating relatively minor variations, an area can feel far removed from another area that uses many common elements.

Find out in advance if your prefab sets will require back-facing polygons. Leaving the back of a prefab hollow guarantees someone will make an environment to display it.

Mirror, Mirror

Designers will love you for creating a flexible set that is painless to work with. Of key importance to achieving that flexibility is making sure your geometry can be mirrored easily on as many axes as possible, so keep a line of symmetry through your work as you go.

Cylindrical structures are an important spot in this regard. It's a good idea to give objects such as pipes a number of sides divisible by four. Initially this may seem odd, but the first time you have a row of pipe sections and try to pivot a seven-sided length of pipe 90 degrees, the logic will become apparent. Seams appear, edges don't line up, and things get messy quickly. With an eight- or 16-sided pipe, you can mirror or rotate an elbow easily without risks of seams.

When working with symmetry in mind, you'll want to restrict any lettering that can't be mirrored to detail pieces. Try to create pieces that work when mirrored vertically as well as horizontally, and consider creating transition pieces that will allow an object to be mirrored to itself for added flexibility.

Origins

You may have a prefab modeled rigidly on a grid, but if your origin is off, it may come into the engine off grid as well. To what degree this happens depends on your particular technology, but be aware of it as you build your meshes in world space. A handy trick for planar wall sections is to place the origin in a corner of the mesh, instead of the center. This allows the wall prefab to be stretched or pivoted around while one corner remains on grid. An origin in the center of a rotated wall leaves both edges off-grid and makes it difficult to line them up again (see Figures 2a and 2b).

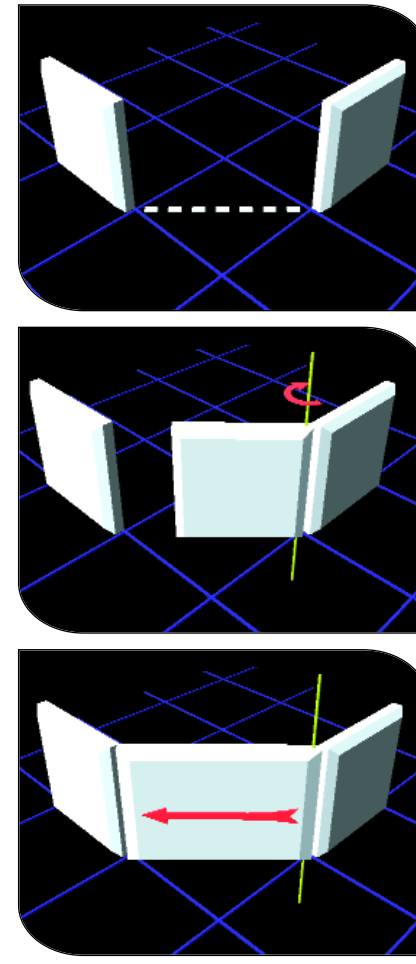


FIGURE 2A. With an origin in the wall's lower corner, a wall can be rotated and stretched to fill an angled gap easily, while the origin stays on-grid.

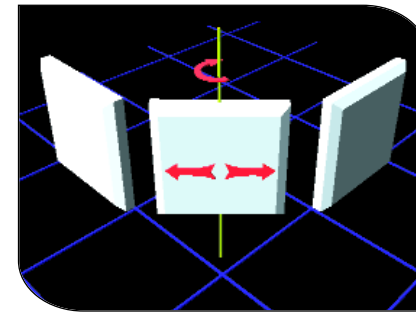


FIGURE 2B. A centered origin results in having to move off the larger grid to cover an angled gap, scaling in multiple directions, and creating seams on both walls, thereby complicating the process and necessitating clean-up.

Don't Forget to Accessorize

The next step in the system is to create a handful of accessory pieces that can be used to break up the appearance of tiling. A castle hallway created from modular wall sections can look entirely unique with the placement of a few busted overhead beam meshes, a worn statue, or loose stones from a collapsed wall. It doesn't take much to fool a player into thinking two essentially identical corridors are unique places; their differences just need to have enough visual impact to count.

A good technique to use when creating accessories is to build them in a further modular fashion. If you're planning to build three different detail statues for the castle in the preceding example, you could separate the statue meshes into a lower section, a torso, and a head. Mix and match those parts, and with a little planning before diving in, your castle can be filled to the parapets with unique statue variants in every hallway.

Prefabs can be very flexible if built in a way that frames rough level geometry. Use a shell system like the one shown in Figure 3 to give otherwise plain geometry a more detailed profile.

You can also create accessories expressly with the intent to conceal various kinds of level work. Your system might match up very well overall, but it's a good idea to have some generic concealment pieces handy. For example, if you're working with a natural-looking rock wall system, there may be instances where the walls are used at odd angles and cause seams. Example concealment pieces might be a rustic beam that can be placed over the intersection, a large sliver of a stone slab, or perhaps some foliage.

FIGURE 3. Modular boundary systems can give simple level geometry great detail while maintaining flexibility.

Custom Pieces

After you've got a base set and enough accessories to embellish the environment, the next step is to create custom areas that the design calls for. It's important that this step come after the basics, as you'll often need information such as how the basic set will transition into the custom pieces. If the piece needs to fit very specifically, you or the designer may consider blocking it in first and using that work as a template for the final geometry.

You can take a bit more freedom from the grid when constructing a made-to-order section of geometry, but you should get back on the grid at the edges of the piece to ensure everything will fit together seamlessly. Also, when building a custom mesh, as with the accessories, ask yourself if there's an easy way to divide it that would allow it to be used more flexibly than just drag-and-drop. For example, if you're creating a detailed starship control room that connects with modular hallways, look for a way to divide the room so an extra section could be added to expand the area. Or perhaps the customized railing that spans the bridge could be broken into sections and used elsewhere.

Level Assembly

Now comes the payoff, designing the level by snapping together the Lego-like units. The particulars of this process will vary greatly from one project to the next, but there are still a few general tips that can help (see Figures 4a–4c).

First, rough in a few ideas before starting construction. Take a blank slate and see what you can do with the pieces you have before going back and requesting more. You might not need that custom piece at all if you can find a clever workaround.

There's nothing wrong with using a piece in a way it wasn't intended. If you create the level simply by snapping together TETRIS unit shaped corridors, the player is likely to reject the modular nature of the environment. It's critical to the workflow's success that users have the ability to improvise and invent new uses for prefabs. Some environment styles are more flexible than others, but there are generally areas of flexibility for any genre. A complicated sci-fi doorway may end up mirrored and used as a window. An altar may be duplicated and used as an archaic column. The possibilities for variation are greatly increased if your particular technology allows for non-uniform scaling of components.

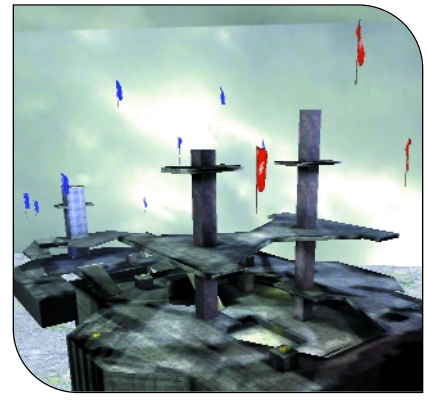
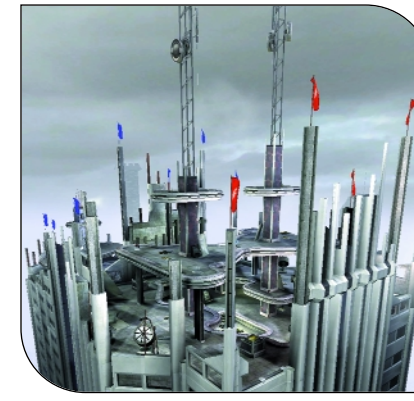
Mix it up even more by using your accessory elements in key construction roles. For instance, you might take a single wooden beam mesh and build a shoddy doorway with it, or take a piece of ductwork and duplicate it in an array to form a unique wall. Each set of geometry presents an array of possibilities.

Finally, if your engine allows for projected textures or lighting, be sure to use them in key spots to further break up repetition of detail throughout the level.

Consider the Benefits

Even if modular construction techniques don't seem like something your project could utilize, there are a few key benefits to consider before making that decision.

First and most importantly, afterthoughts are rampant and a fact of life in game development. Working with prefabs offers an unprecedented degree of freedom to go back and edit an original piece, while having every instance of that piece automatically update throughout your entire world. Don't like the texture on that monitor? Implement it for now, and change it when you have the time. In many engines, an entire piece of geometry could be replaced as long as the new one fits the function of the old piece reasonably well. Working with movable

FIGURES 4A–4C (left to right). A small selection of instanced prefabs (left) can turn a basic level (center) into a far more complex world (right).

chunks of geometry also allows for easy revisions on the designer's end, without having to go back and consult with artists. Are testers telling you they need an alternative exit from that room? That's no problem in a modular environment.

Regardless of rendering power, memory will always be a concern. Working modular generally allows an engine to work with many instances of the same object. A complicated wall piece can be duplicated many times with minimal memory impact after the initial placement. Even the densest-detail environment can get away with using only a handful of meshes in memory.

Another important consideration is consistency, and with bigger teams it's a bigger concern. By using a prefab set that has been built with minimal cooks in the kitchen, even a sprawling, robust level will maintain a style throughout. Such consistency is far more difficult to achieve in an entire world built of unique geometry.

We all want to reach as many players as possible with our games, and a modular system allows you to create very scalable levels of detail (LODs) for different platform specifications. Developers can flag high-detail extra touches so they won't draw on various video settings. If your technology allows or requires LODs, you'll find a modular construction technique facilitates that very well.

In addition, modularity allows team members to concentrate on doing what they are best at. Level designers with strong gameplay skills needn't worry about creating the loads of details required in high-end graphics. With a modular construction set they can focus on laying out the game and not get bogged down trying to create what should be considered art assets.

Finally, working with prefabs can help greatly if you plan on releasing a game-editing utility. Titles such as MORROWIND, NEVERWINTER NIGHTS, the WARCRAFT series, and countless past titles that have used modular or tile-based techniques have made their editability a key selling point. Editability helps extend a title's shelf life and creates a stronger user community.

Limitations and Drawbacks

Developers on some projects may see the word "modular" and panic. Initially, old-school, graph-paper-designed *Dungeons & Dragons* levels come to mind. Ninety-degree hallway corners and evenly spaced doorways creep into their nightmares.

The bottom line is that modular construction is indeed more restrictive than having a staff of dozens create an entire level as custom geometry and unique artwork. But for most teams, that kind of content creation workflow just isn't feasible. The key point is that having to fix some obvious tiling along the way is still far easier than creating a game full of custom content. Based on your game and its goals, you can decide the right level of modularity for it. If you're nervous, try starting on a smaller scale of prefabs that will allow for more detail in a smaller area.

Some designers find it incredibly stifling to work with geometry they didn't create. This can be a common scenario, but as graphics continue to improve at a startling rate, those designers will eventually have to accept that level-editing tools are not going to be able to create the kind of detail we'll be seeing in the next few years of AAA titles. It's really no different from using textures created by someone else. After getting accustomed to it, it's hard to ignore the benefits of being able to see one's gameplay ideas realized quickly.

Farther Down the Road

Looking farther into the future of game environment creation, who knows what we may encounter? Perhaps modular techniques may lead to truly immersive procedural worlds. Some say that's inevitable to some degree, and others aren't happy about it and want to resist moving in that direction. Wherever we all end up, and regardless of the specific nature of your current project, you're using modular construction in some fashion already. If your artists are creating tiling textures, if you have a standard step height, or if you have a tile-based terrain system, you're already in the mentality of building in a modular way; you merely need to extrude it into the rest of the environment.

brian hawkins

Creating an Event-Driven Cinematic Camera: Part 2

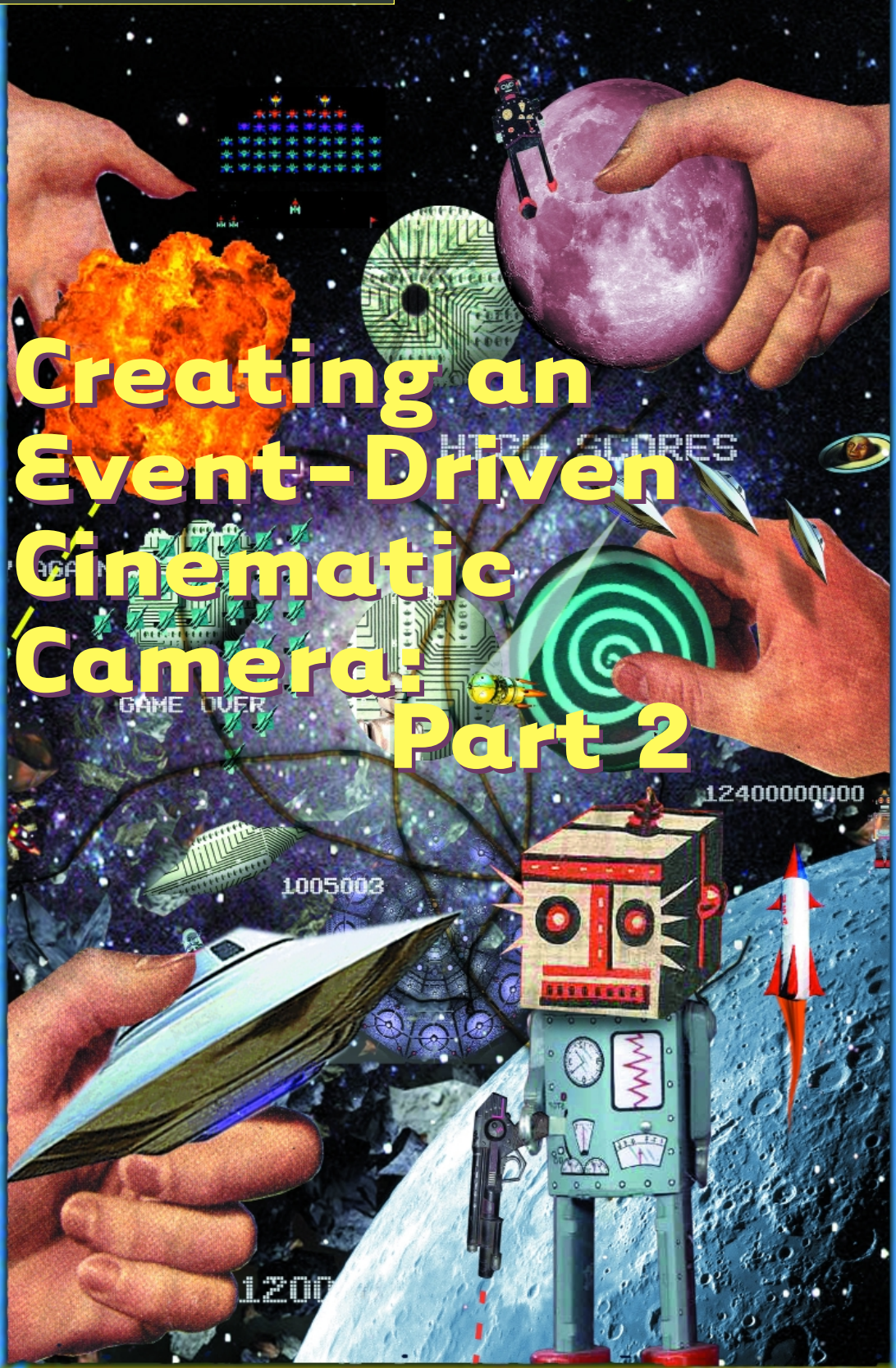


Illustration by Steve Munday

Once upon a time, it was a death wish for a game to be based on a movie license. However, things have changed considerably in recent years. There have been a number of well done and successful game titles based on movies, and on the flip side there have been several movies released that had games as their origin. With the crossover between movies and games finally starting to show some success, it is time to revisit how Hollywood can actually be helpful to the game industry.

In the past century, motion pictures have developed a visual language that enhances the storytelling experience. Equally important, audiences have grown accustomed to certain conventions used to tell these visual stories. Unfortunately, very little of this knowledge has been translated for use in interactive storytelling.

Last month, in Part 1 of this two-part series, we looked at how to describe a cinematic camera shot in general terms so that it could be automatically converted to camera position and orientation within the game. To conclude, this month's article brings it all together by presenting a system that can choose the best shots and connect them together. Once finished, these concepts can be joined to form a complete basis for a cinematic experience that improves the interactive storytelling of games by giving players access to the action within a game in ways that make sense to them instinctively.

Film Crew

Major motion pictures are made by hundreds of different people all working together in a huge team effort. To transfer the cinematic experience to the world of games, we can take certain established, key roles from the film industry and translate them into entities in the computer. Objects in object-oriented languages such as C++ can conveniently represent these entities. In this article, we will look at the three primary roles and describe their responsibilities as objects. From this, you can build architectures to coordinate real-time cinematic camera displays. Before going into detail about each role, let's take a brief look at each in turn.

The first job belongs to the director. In films, the director controls the scene and actors to achieve the desired camera shots that will then be edited later. However, because our director object will have little or no control over the game world, this responsibility shifts to determining where good camera shots are available and how to take advantage of them.

Once these possibilities are collected, they are passed on to the editor who must decide which shots to use. Unlike in motion pictures, however, the editor object must do this in real time as each previous shot comes to an end. The editor is also responsible for choosing how to transition between shots.

Finally, once the shot and transition have been decided upon, it becomes the cinematographer object's task to transform that information into actual camera position and movement within the game world. With this basic idea of how all the roles and responsibilities fit together, we can move on to a closer look at each individual role.

BRIAN HAWKINS | *Brian began his career doing research at Justsystem Pittsburgh Research Center, where he focused on scripted character animation using natural language. He worked at Activision as the game core lead on* Star Trek: Armada, *and contributed to* Civilization: Call to Power *and* Call to Power 2. *His last project was working for Seven Studios as lead programmer on* Defender. *Brian holds a B.S. in mathematics and computer science from Carnegie Mellon University.*

Through the Viewfinder: The Director

As mentioned previously, the director's role in the game world is to collect information on available shots and their suitability for inclusion in the final display. This is the one place where human intervention is necessary, after which no more human input is necessary. It is currently impossible to create a system sophisticated enough to determine the priority of events within the game world from a creative standpoint.

Instead, programmers and scripters are given the ability to provide information about priority and layout of interesting events, hence the term used in this article — event-driven cinematic camera, through a `suggestShot` method on the director object. This information will then be used by the editor for a final decision on which shots to include. Following is a breakdown of the information necessary to make these decisions.

The first and most important piece of information is the priority of the shot. The priority represents how interesting a particular shot is compared to other shots available at the time. Thus the value of priority is relative, which means there is no definitive meaning for any particular number. You must therefore be careful to remain consistent within a single game in order to give the priority levels meaning. For example, all other values being equal, a shot with a priority of two is twice as interesting as a shot with a priority of one.

The second piece of information required is the timing of the shot. Timing is the most complex part of the editing process, and the sooner an event can be predicted, the better choices the editor can make. Timing breaks down into four values: start time, estimated length, decay rate, and end condition. The start time is obviously the beginning of the event. The estimated length is a best guess at how long the shot will last. The decay rate determines how quickly the priority decays once the event begins. Finally, the end condition determines when the shot should be terminated. Let's look at decay rate and end conditions in more detail.

The decay rate is used to determine the actual priority at a given time t using the starting priority p and a constant, k. The constant is provided as part of the decay rate information, since

it will differ from shot to shot. The other information for decay rate is the equation to use for determining the actual priority. For maximum flexibility, this should be a function object that takes t, p, k, and the start time, t_s, and returns the priority for that time. Two useful functions that should be predefined as function objects for this parameter are:

$$p(t) = p(1 - k(t - t_s))$$

$$p(t) = p((1 - k(t - t_s))^3)$$

These functions should suffice for most circumstances. Notice that the second equation cubes the value rather than squaring it. This is important, because it ensures that the priority remains negative after a certain amount of time has passed, whereas squaring would have caused the result to always remain positive. Figure 1 shows the resulting graphs of these functions as a visual aid for understanding how decay rate affects priority.

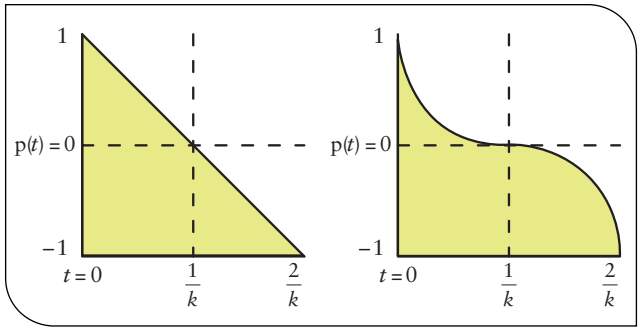


FIGURE 1. Decay rate graphs, showing how decay rate affects shot priority.

The end condition is best specified as a function object that returns one of three values. The first value indicates the shot cannot be terminated yet, the second value indicates the shot can be terminated if another shot is ready, and the third value indicates that the shot must be terminated. The reason for the middle value is that it gives the editor more flexibility in choosing shots by allowing a choice of new shots within a certain time, rather than instantaneously when the shot is terminated.

Next comes the shot information. This is all the information needed by the cinematographer to change the shot from a suggestion into a real in-game shot. This includes information such as the primary actor and secondary actor, if any. In addition, the shot size, emphasis, angle, and height may be necessary. Refer to last month's article for more information on determining this information as well as the following scene information.

The scene information consists of the actors within the given scene and the current line of action for that scene. Unfortunately, scene information can change dynamically as actors move around and the cinematographer changes the line of action. Because of this fact, it is best to store the scene as a reference through the primary actor of the shot that is being suggested.

The director's other responsibilities are to provide the editor with a list of currently available shots at any time and to ensure that this list is up-to-date. Keeping the list up-to-date primarily involves removing shots that are no longer valid. A shot becomes invalid when the priority modified by decay rate, as discussed previously, falls below zero. Once the editor chooses a shot, it is also removed from the list of shots. This brings us to a discussion of how the editor chooses a shot.

Slice and Dice: The Editor

The editor is responsible for choosing the next shot that will be shown as well as any transitions between shots. First, let's look at the process of choosing the next shot. The majority of the information needed is provided with the shot suggestions from the director, but there are parameters that can be used to give each editor its own style. The two parameters involved in shot decisions are the desired shot length, l_{shot}, and the desired scene length, l_{scene}. By setting these for different editors, the shots chosen will vary for the same set of circumstances. For example, one editor could prefer short scenes filled with one or two long shots by setting the shot time and the scene time to be relatively close values. On the other hand, another editor could prefer longer scenes filled with short shots. This provides a number of options when choosing an editor for a particular situation.

The time for choosing the next shot is determined by a change in the return value of the end condition for the current shot. Once the current shot indicates that it can be terminated, the editor must obtain the list of currently available shots from the director. From this list, the editor object then filters out any shots whose start time is too far in the future. If the end condition requires immediate termination, this excludes all shot suggestions whose start time is not the current time or whose start time has not already passed. Otherwise, all shots whose start time is no more than l_{shot} beyond the current time are considered.

To choose the shot from this list, we must sort them based on a value that represents the quality of each shot suggestion and then take the shot with the highest value. Before we can compute this value, we need to introduce a few other values that will be used in its calculation. First, we consider the desired shot length versus the estimated shot length, $l_{estimated}$:

$$p_{length} = \begin{cases} 2l_{estimated} - l_{shot} & \text{if } l_{estimated} > l_{shot} \\ l_{shot} & \text{if } l_{estimated} \le l_{shot} \end{cases}$$

Then we look to see if the actors have any relation to those in the last shot:

$$c_{actor} = \begin{cases} 4 & \text{if both actors are the same} \\ 3 & \text{if one actor is the same at the same priority (primary versus secondary)} \\ 2 & \text{if one actor is the same at different priority} \\ 1 & \text{if both actors are different} \end{cases}$$

Next, we check to see if the new scene matches the old scene. For this the editor must also keep track of the time spent in the current scene, t_{scene}:

$$c_{scene} = \begin{cases} 1 & \text{if current scene} \neq \text{new scene} \\ l_{scene} - t_{scene} + 1 & \text{if current scene} = \text{new scene} \end{cases}$$

Finally, the priority is modified by the decay rate discussed earlier if the shot event has already commenced:

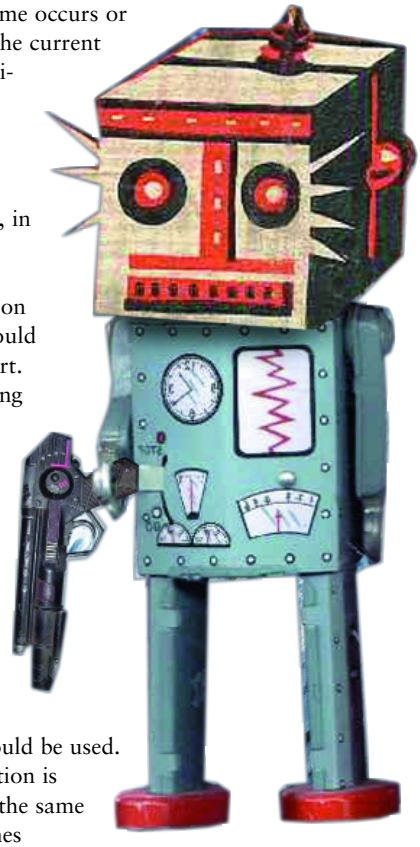
$$p_\Omega(t) = \begin{cases} p & \text{if } t < t_s \\ p(t) & \text{if } t \geq t_s \end{cases}$$

Once we have all this information, we can compute the quality value of each shot on the list:

$$q(t) = p_\Omega(t) \cdot c_{scene} \cdot c_{actor} \cdot p_{length}$$

Notice that the values c_{actor} and c_{scene} allow us to maintain consistency for our shots. This is a very important property of good film directing and editing and should not be overlooked in interactive cinematography, even though it is more difficult to maintain.

You may have also noticed that when calculating $p_\Omega(t)$ that t can be before t_s, thus it is possible under some circumstances to choose a shot that has not started yet. In this case, we hold on to the shot and wait for one of two events: either the shot start time occurs or the end condition of the current shot forces us to terminate. Upon the occurrence of either event, we must once again test to see which is the best shot, in case a better shot has come along or we are being forced to move on before the shot we would like to display can start.



Now that an ordering exists that allows us to choose the next shot, the only remaining choice necessary is the transition from the current shot to the new shot. If we are transitioning between different scenes, the choice is easy, a cut or fade should be used. However, if the transition is between two shots in the same scene, the logic becomes slightly more complex. Within a scene it is important to maintain the line of action; in other words, to keep the camera on one side of a plane defined for the scene so as not to confuse the viewer's perception of scene orientation.

Let's consider the various permutations that can occur between shots and what type of transition should be used. For now, we will break them into fading (think of cutting as an instantaneous fade) and camera movement.We will go into more detail on moving the camera later. First, if the actors remain the same between the shots, then we can preserve the line of action and use a fade. Likewise, even if the actors change but the new line of action lies on the same side of the line of action as the previous camera position, then a fade can still be used.

However, if the two lines of action differ significantly, then a camera move needs to be performed. The camera move should allow the line of action to change without confusing the viewer. To get a rough approximation of the distance the camera must travel, compare the distances between the current and new camera positions and the current and new focal points. Now compute how fast the camera must move to travel that distance in

$$s = \frac{\left|\max(\Delta_c, \Delta_f)\right|}{\{t : p(t) = 0 \wedge t > 0\}}$$

the time it would take for the new shot to become uninteresting:

Where Δ_c is the vector between camera positions, Δ_f is the vector between focal points, and $p(t)$ is the priority decay formula for the shot.

If the camera move cannot be made at a reasonable speed, then a new shot must be chosen, unless the actors from the last shot would not be visible in the pending shot. Otherwise, a new shot should be chosen with preference for close-ups that include only one actor, thus making the next transition easier. We can now move on to realizing the shot and transition. For the decay formulas given earlier, t would be $t_{start} + 1/k$.

Lights, Camera, Action: The Cinematographer

Last month, we covered the math necessary to turn a description of a shot into actual camera position and orientation. This month, we will build on that and flesh out the role of the cinematographer by covering the handling of transitions.

The simplest transition is the cut, where we only need to change the camera position and orientation to a new position and orientation. Only slightly more complex is the fade, which provides a two-dimensional visual effect between two camera views. When fading, it is important to decide whether to fade between two static images or allow the world to continue to simulate while the fade occurs. Allowing continued simulation implies rendering two scenes per frame but eliminates the need for pauses in gameplay. If you are able to handle the extra rendering, interesting fade patterns can be achieved by using complementary masks when rendering each scene. Depending on the hardware available for rendering, you may only be able to do black and white masks, or you could go all the way to

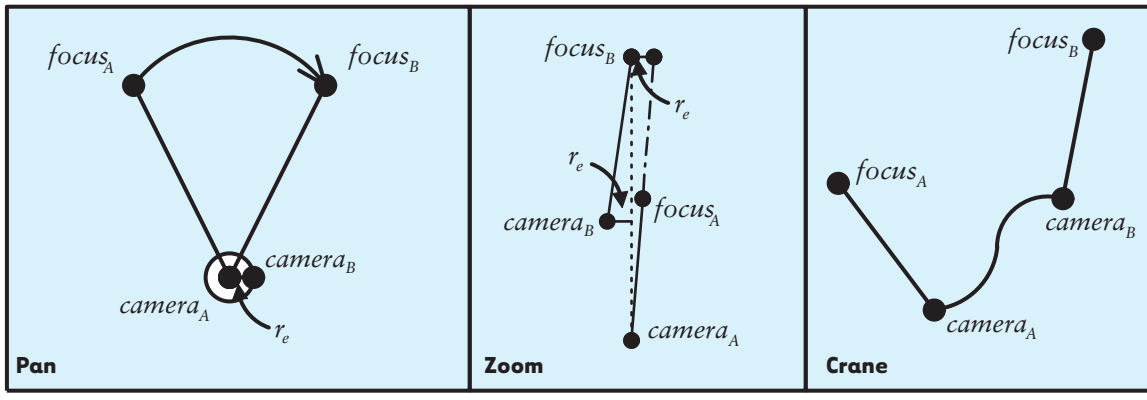


FIGURE 2. Shot transition criteria, where r_e is the radius of acceptable error.

alpha-value masks.

The other group of transitions involves moving the camera. The three transitions we will consider are pan, zoom, and crane. The decision of which move to make depends on the camera and focal positions for the two shots. Figure 2 shows the various situations that lead to the choice of a particular shot. The pan is used if the camera is in approximately the same location for both shots and only the focal point moves. Though this happens rarely in an interactive environment, when it does happen the old camera position can be kept and only the orientation needs to be animated to the new orientation. Similarly, the conditions for zooming are fairly uncommon, as both the camera positions and focal points must lie close to the same line, but when it does occur the camera field-of-view can be used to allow a much more interesting transition than a simple camera move.

Finally, we come to the most complex transition, the crane. The best method for creating a crane move is often by borrowing the services of the AI's path-planning algorithm in order to avoid moving the camera through objects. It is best if the path planning also handles orientation, as this will lead to better results than interpolating between the focal points.

Unfortunately, getting crane shots to look their best is a complex process for which this is only a starting point. If you do not have the time to invest in making them work, you may wish to leave them out altogether.

Beyond the Basics

You now have enough information to create your own basic cinematic system to include in your game. There is plenty of room to go beyond this basic system. Research on some of these areas has already been conducted in academic circles. For instance, events that involve conversations between characters could be specified as a single suggestion rather than manually suggesting each individual shot during the discourse. "The Virtual Cinematographer" and "Real-time Cinematic Camera Control for Interactive Narratives" (see For More Information) describe how director styles can be created to specify camera shots automatically for these situations. This reduces human involvement, which is always important as it allows other features to be added to the game.

Another important aspect of cinematography that is only now becoming possible with the power of newer graphics hardware is depth-of-field. This is often used as a mechanism to draw attention to various elements in a scene of a film. As rendering of depth-of-field becomes more common, it will be important to develop controls for it that are based on the principles learned from cinematography. It is even possible to extend the concept of depth-of-field in ways that would be difficult in real-world filmmaking. "Semantic Depth of Field" in For More Information talks about selective application of depth-of-field effects on important elements of an image.

As you can see, there is a wealth of information out there and plenty of room for experimentation and new ideas. As games continue to grow in popularity, they must meet the demands of the more general audience that is used to the conventions of films. There is much to do in order to reach this goal and continue to expand the scope of game development. Continued innovation and experimentation in this area will bring out greater variety of expression on the part of game developers, and richer, more compelling

FOR MORE INFORMATION

Amerson, Daniel, and Shaun Kime. "Real-time Cinematic Camera Control for Interactive Narratives." American Association for Artificial Intelligence, 2000. pp. 1–4.

Arijon, Daniel. *Grammar of the Film Language*. Los Angeles: Silman-James Press, 1976.

He, Li-wei, Michael F. Cohen, and David H. Salesin. "The Virtual Cinematographer: A Paradigm for Automatic Real-Time Camera Control and Directing." *Proceedings of SIGGRAPH 1996*. pp. 217–224.

Katz, Steven D. *Film Directing Shot by Shot*. Studio City, Calif.: Michael Wiese Productions, 1991.

Kosara, Robert, Silvia Miksch, and Helwig Hauser. "Semantic Depth of Field." *Proceedings of the IEEE Symposium on Information Visualization 2001*.

Lander, Jeff. "Lights… Camera… Let's Have Some Action Already!" Graphic Content, *Game Developer* vol. 7, no. 4 (April 2000): pp. 15–20.

BioWare's NEVERWINTER NIGHTS

GAME DATA

PUBLISHER: Infogrames
NUMBER OF FULL-TIME DEVELOPERS: 75 at peak, representing approximately 160 man-years of development
NUMBER OF EXTERNAL STAFF AND CONTRACTORS: Approx. 40 QA testers at Infogrames, 5 sound contractors, and 20 translators
PROJECT LENGTH: Approx. 5 years
RELEASE DATE: June 2002
PLATFORM: PC, with Mac and Linux clients forthcoming
AVG. DEVELOPMENT HARDWARE USED: P3-600MHz to P4-2000MHz with GeForce 3s, 512MB RAM, and 30GB hard drives.
DEVELOPMENT SOFTWARE USED: Visual Studio C++, 3DS Max 3 & 4, Adobe Photoshop
NOTABLE TECHNOLOGIES: Bink, Miles Sound System, Gamespy, BioWare Aurora Engine, BioWare Neverwinter Aurora Toolset

NEVERWINTER NIGHTS (NWN) was conceived in 1997 as the ultimate pen-and-paper role-playing game simulation. BioWare's goal with the project was to try to capture the subtleties of a pen-and-paper role-playing session in a computer game, including a fully featured Dungeon Master with full control over the game world as it unfolds, and an extremely approachable toolset to allow nontechnical users to make basic content.

Early in BioWare's development of BALDUR'S GATE it became clear to us how the evolution of the role-playing game genre would unfold. We saw the explosion of fan-created content for first-person shooters and we rationalized that the role-playing genre was ready for a similar renaissance. It was going to take a lot of work to do it right, but even near the project's completion, we realized that at the start we had greatly underestimated the effort it would take to complete a project of this size.

NEVERWINTER NIGHTS was also inspired by the early massively multiplayer games like ULTIMA ONLINE. Our experience online was that we had the most fun when we were adventuring with a moderate-sized group of friends, with a Game Master creating an adventure for us in real time. This experience was one of the foundations of what we wanted to capture in NEVERWINTER NIGHTS.

NEVERWINTER NIGHTS was the largest and most ambitious project BioWare has yet undertaken, beyond the 250-hour BALDUR'S GATE II: SHADOWS OF AMN. Our goal was to create a game with significant impact, and also to deliver on all of our goals, not just a couple. As a result we had an extremely large team working on NEVERWINTER NIGHTS.

At its peak, the team numbered more than 75 people — with 22 programmers working on aspects as diverse as the game client, independent servers, the Dungeon Master client, and the world creation tools (the BioWare Aurora Neverwinter Toolset). Not only did the final game feature a large number of programmed features, but we also had hundreds of monsters, thousands of custom scripts, and a substantial single- or multiplayer campaign (featuring 60 to 100 hours of gameplay). Coordination of such a large team presented us with a number of unique management challenges, and in retrospect we learned a number of lessons regarding managing huge projects, many of which are described in this article.

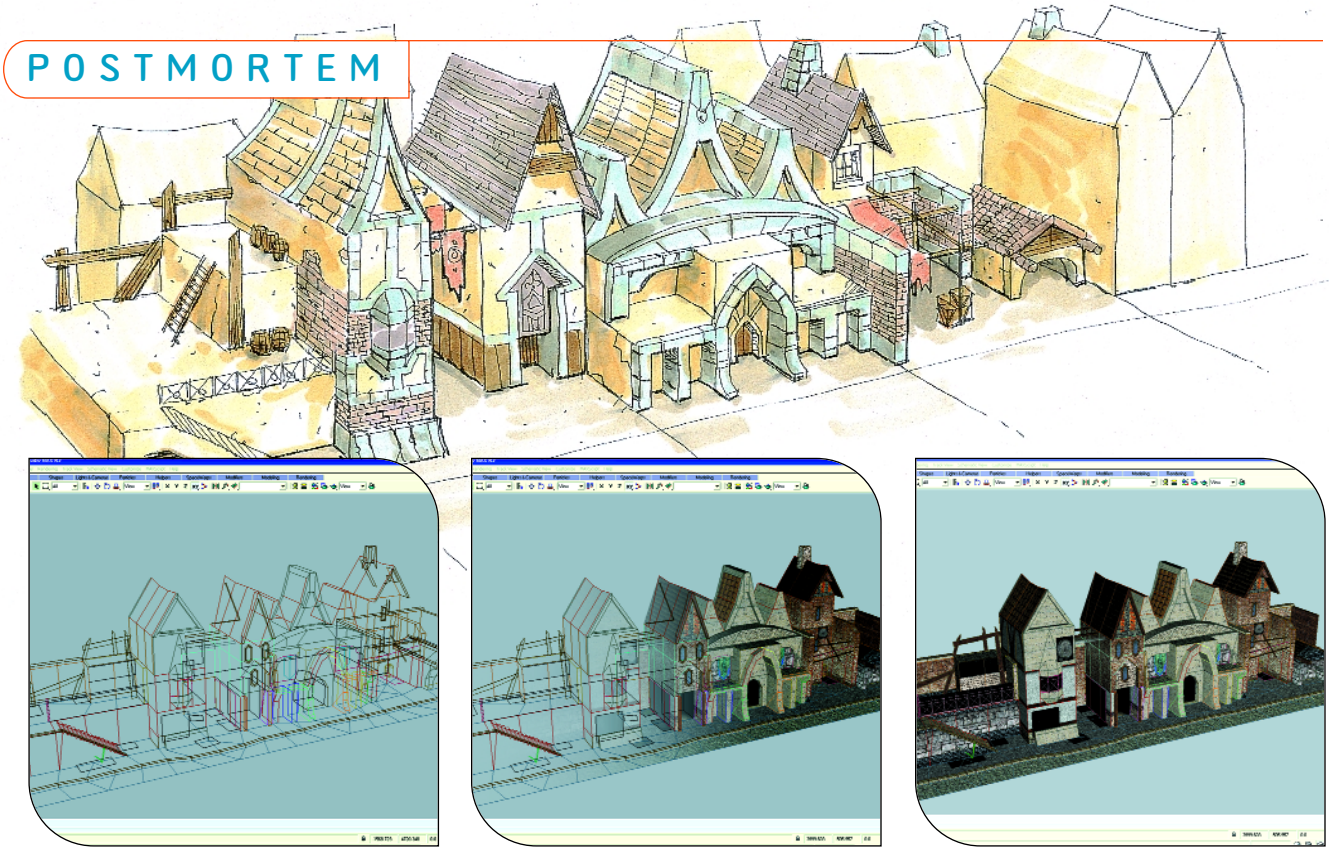
SCOTT GREIG | *Scott is director of programming at BioWare and was lead programmer on NEVERWINTER NIGHTS.*
RAY MUZYKA | *Ray is joint-CEO of BioWare and NEVERWINTER NIGHTS' co–executive producer.*
JAMES OHLEN | *James is BioWare's director of writing and design and was co–lead designer on NEVERWINTER NIGHTS.*
TRENT OSTER | *Trent was NEVERWINTER NIGHTS' project director and producer.*
GREG ZESCHUK | *Greg is joint-CEO of BioWare and was co–executive producer on NEVERWINTER NIGHTS.*

What Went Right

1. Constant communication. With the NWN team growing at the end to upwards of 75 people, communication of the goals of the project, and the day-to-day development decisions, became a critical necessity. We formed a tight communication network, with the leads in each area summarizing daily challenges, potential pitfalls, and areas of concern. Each significant code change required consultation with the various people responsible for the systems that would be touched — a fundamental change required a quick meeting between three to five people simply to make sure everyone who was affected would be aware of how the change would affect them. While this might superficially seem like an inefficient way to work, it did result in a number of benefits for the project and the team. First, the constant communication to achieve difficult goals brought the entire team together, and when difficult problems arose there were always a few knowledgeable people with a familiarity with the problem that could be consulted.

Furthermore, NWN was painstakingly documented, from technical design docs and art style guidelines to rules and level design documents. The team leads created and updated massive schedule documents detailing every aspect of the project. The tools group designed and created a project manager program. This tool facilitated faster and more organized communication between the departments. While documents would sometimes fall out of date, it was still a big step forward compared to our previous projects.

While documentation to this level is probably not required (and might even prove to be a hindrance to progress) on many games, on a role-playing project of



A set of Neverwinter City tiles progressing from concept to completion.

the size of NWN, it was critical. With any large project, one of the major challenges is making sure the team works as a unit to achieve a common goal, rather than a number of parallel but unrelated goals. We found that the style and manner of communication on the NEVERWINTER NIGHTS team was instrumental in both building the team spirit and making sure the game was successful.

2. Extensive tool effort. Compared to BALDUR'S GATE, NEVERWINTER NIGHTS allocated five times the manpower to making the game-creation tools suite. Although part of the increase was due to the fact that the Neverwinter Toolset was to be publicly released, we would have still invested fairly heavily into the tools even if they were only intended for internal use. The decision to go forward with a new tool, or a new feature for an existing tool, depends on whether the time required to make the tool will be made up in time saved by using the tool.

In BioWare's past projects, we have generally found that if a task that could be automated by a tool was going to be performed more than once, then the tool saved time in the long run. Despite problems inherent in using a tool under development, having a large and robust toolset allowed for very rapid implementation of design and art content. We ramped up our tools department during NWN's development, and it has served the company well to have a group that can service the tools, database, and installer needs of the entire company.

3. Multiplayer integration from the outset. Although BALDUR'S GATE was intended to have multiplayer support from the beginning, we did not actually start programming the multiplayer systems until relatively late in that project. As a result, some of the multiplayer aspects in BALDUR'S GATE — such as forcing all players to see all dialogue — were less than optimal.

In NEVERWINTER NIGHTS, the multiplayer systems were integrated directly into the original design. Even in single-player, the game acts like a multiplayer game with a single client attached. Although this deep integration increased the time to develop each system (compared to a single-player-only system), it did result in an overall reduction in the time required to integrate multiplayer and ensured that all the systems were optimized for multiplayer play.

One useful lesson from both the BALDUR'S GATE series and NEVERWINTER NIGHTS was how much time QA testing of a multiplayer game takes compared with testing just the single-player game. We have found that three to five times as much testing is needed for multiplayer role-playing games compared with single-player. Thus we require 30 to 50 testers (including both on-site and external testers) on our multiplayer projects for three- to six-month periods — not a small undertaking.

4. Experienced team members focused on quality. Having numerous BioWare veterans on the NEVERWINTER NIGHTS team was crucial to holding the project together and ensuring the development efforts were successful. We hired a number of new people during the course of the game, practically all of whom had no prior game development experience, but we were very fortunate that a number of people that had worked on the BALDUR'S GATE series also worked key roles on NWN. Their RPG development experience served as the cement that held everything together on the project and enabled them to circumvent many of the pitfalls typically encountered when developing a story-based role-playing game. In addition, their ability to mentor

new hires was essential in building a strong team, both for NEVERWINTER NIGHTS and for BioWare.

Even though the majority of the team members were not experienced game developers, after they joined the team they had access to mentors to help them learn their craft. BioWare's culture — based on a matrix structure with departments of programming, art, QA, and design — encourages learning and aggressive transfer of knowledge, which we believe is the best foundation for building a strong development team.

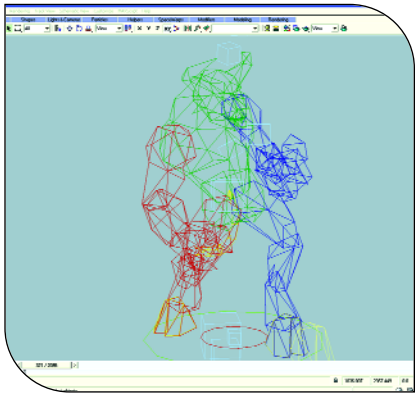
Many of the core team members worked on the project for a number of years — the entire duration of the project from the idea stage to completion was slightly more than five years. While we pushed aggressively through the entire development, there was never a sense the game would be shipped before it was ready. We set out to achieve all of our goals, and we never wavered from that plan, even during some of the complicated issues that arose in the project moving from Interplay to Infogrames. Fortunately, Infogrames was able to come onto the project late in its lifespan and mobilize the resources required to ensure the quality of the game by our intended launch date.

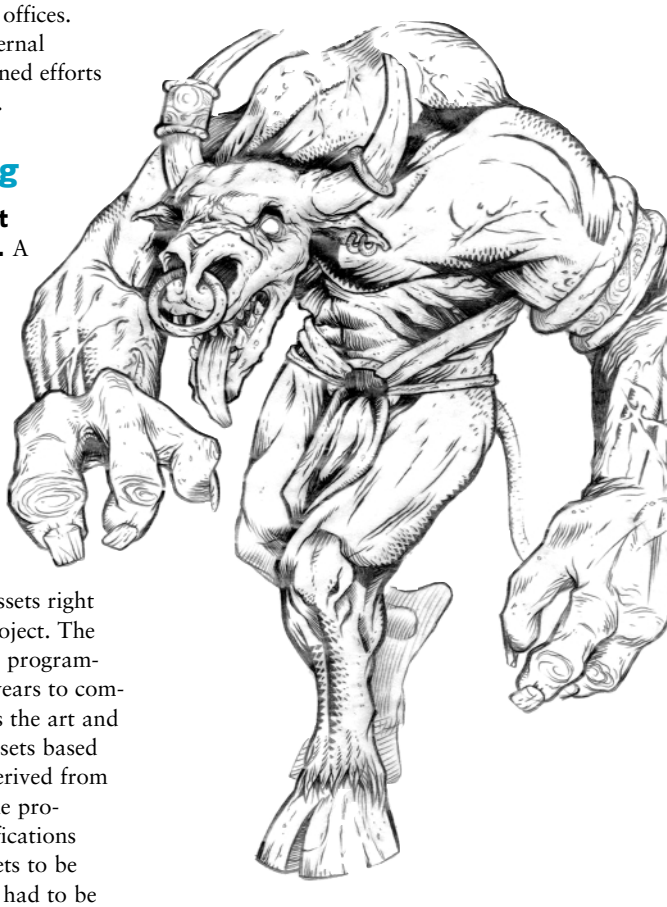
5. Sharing resources with other projects. BioWare relies heavily on our ability to draw upon manpower from the rest of the company to help out on a project in the final stages of production. All of our projects have done this in the past, and NWN was no exception. Designers, artists, and programmers came on from the old Infinity engine team as soon as BALDUR'S GATE II: THRONE OF BHAAL was completed. Many of these people were responsible for key aspects of the project, in the same way that many of the NWN team members had been responsible for systems in MDK2 and BALDUR'S GATE II. The development teams weren't the only people who helped out at the end; systems administrators, front-office staff, and the PR department also helped test the game.

Even with the help of a lot of people from BioWare, finishing a game the size of NEVERWINTER NIGHTS was a huge undertaking — in addition to our 75-person team working on the game at BioWare, we had 10 on-site testers from Infogrames at our office, eight German and five Korean translators in-office in the last three months of development doing simultaneous translation of the game, and more than 35 external testers between Infogrames' various offices. Coordinating all of these external resources required the combined efforts of five producers at BioWare.

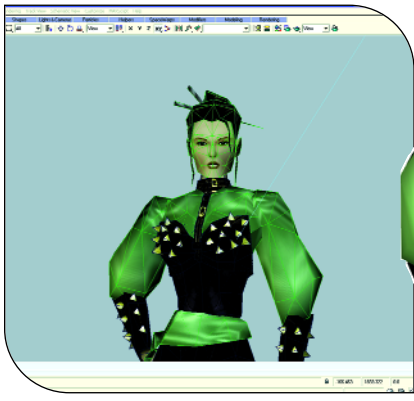
What Went Wrong

1. Resources added at non-optimal times. A large RPG such as BALDUR'S GATE or NEVERWINTER NIGHTS requires a similarly large amount of art, design, and programming resources. One of the problems that we encountered was what to do while the new game engine technology was being developed. Due to our schedule, we needed to start working on art and design assets right from the beginning of the project. The problem was that it took the programming team three and a half years to complete the game systems. Thus the art and design teams had to make assets based on technical specifications derived from early prototypes. As the game progressed, many of these specifications changed, requiring some assets to be rebuilt, or else workarounds had to be adapted in the game code to allow for old and newer assets to work together.





This detailed Minotaur concept was built, textured, and animated in 3DS Max.



Sedos Sebile, a character from NEVERWINTER NIGHTS' plot, standing larger than life.

In an ideal world, the length of the project would have been longer, with the programming done at the beginning with only a skeleton team of artists and designers to provide prototypes. Full production would have then gotten underway once the engine was complete. Unfortunately, this was not feasible due to schedule limitations and interproject scheduling pressures. We have found that when we are reusing or building on an existing engine framework, art and design can be completed with little risk of having to rework resources — problems like we encountered on NWN seem to occur mainly when we are creating a new engine from scratch (we encountered similar issues during the creation of BALDUR'S GATE, for example, but not during the various BG derivatives), and we are keeping this in mind as we schedule new projects in the future.

2. Incomplete prototypes. Even though we put a lot of effort into prototyping important game systems, on some occasions we completed what we thought were full-featured prototypes of major game systems only to find out later that they didn't address a number of important issues. In our haste to get into full production on NEVERWINTER NIGHTS, we didn't properly analyze all of the questions that needed to be addressed by the prototypes. This resulted in spending time late in the development cycle sorting out problems with key systems of the game.

Development of our new game engine was an extremely long process, so some of the initial prototyping lessons were forgotten or inadequately documented. In some cases we didn't thoroughly review our original goals when implementing features later in the project.

As with any new-engine game, there was too little time available to prototype gameplay. Our prototypes focused instead on technology and the individual features of the game. While this kind of prototyping was important, it would have been very useful to have early feedback on how the game played, particularly with regard to the interface and story line.

Because NWN was a rule-based game, and rules implementation was at the end of the schedule, we were only able to test actual gameplay near the end of the development cycle. Due to our inability to prototype a number of design components, we ended up reworking them. As a result we plan to prototype story lines in future games earlier in the development cycle. One of the ways we plan to do this is to reuse the BioWare Aurora Toolset as a rapid prototyping tool for story design, even for games with radically different interfaces and rules systems.

3. Delayed rule implementation (including tools implementation delay). The *Dungeons & Dragons* rules system in NWN was implemented according to a priority system established by the leads on the project. We failed to take into account how some minor addition to the rules system could have far-reaching effects throughout the game. This forced the designers through an aggressive series of revisions to the areas and characters in the official campaign story. In the end, we were able to tune the game appropriately, but we put the level designers, scripters, and writers through a very trying period.

The delay in rules implementation caused a ripple effect with the tool development. Furthermore, we reworked the tool interface late in the project to make it more approachable to nontechnical developers. This rework had a significant effect on the ability of our designers to finish off content, since they were using the exact same tools to fix bugs and finish up the game's development.

4. Late feature additions; innovation for its own sake. To ship a game that takes five years to develop takes a fair amount of intestinal fortitude. You really can't second-guess your decisions or you'll have no chance of ever completing the project, so the leads of the project agonized over some late feature additions to NEVERWINTER NIGHTS. Given that the game was in development for such a long period, we were all concerned it might look dated by release. To combat this issue we laid out a plan to add a number of high-impact but relatively easy-to-implement features late in the development cycle to improve the game's visual

quality. These additions resulted in constant concern among the artists who had to generate the new art required to support the late-added technologies. In the end it all worked out because of large personal efforts by many team members.

From the start there was a strong desire to make NWN a unique game distinct from the BALDUR'S GATE experience. While this did lead to the development of new systems that were better than those of BALDUR'S GATE, it also led to an excessive amount of time spent on design and prototyping of features that ultimately could not be implemented. We'd often sink a considerable amount of research into creating an innovative system, only to fall back on a similar system that worked better in the earlier Infinity engine.

Too often we were determined to start at square one, instead of expanding on what had worked with our previous games. We learned that it is important to choose our battles. In the future, when designing a game set in a genre that we have experience with, we will look more closely at what has worked well previously and aim to innovate only in the areas of our past games that our fans and critics perceived as weak.

5. A lot of demos. A side effect of the attention NEVERWINTER NIGHTS received in the years prior to its release was that we built a number of demos for trade shows and press visits, more than typically occur for most major releases. We probably announced the game too early in its development cycle, and it took a long time to complete the game with the promised feature set. Each time we built a demo there was an impact on the team in terms of both focus as well as schedule.

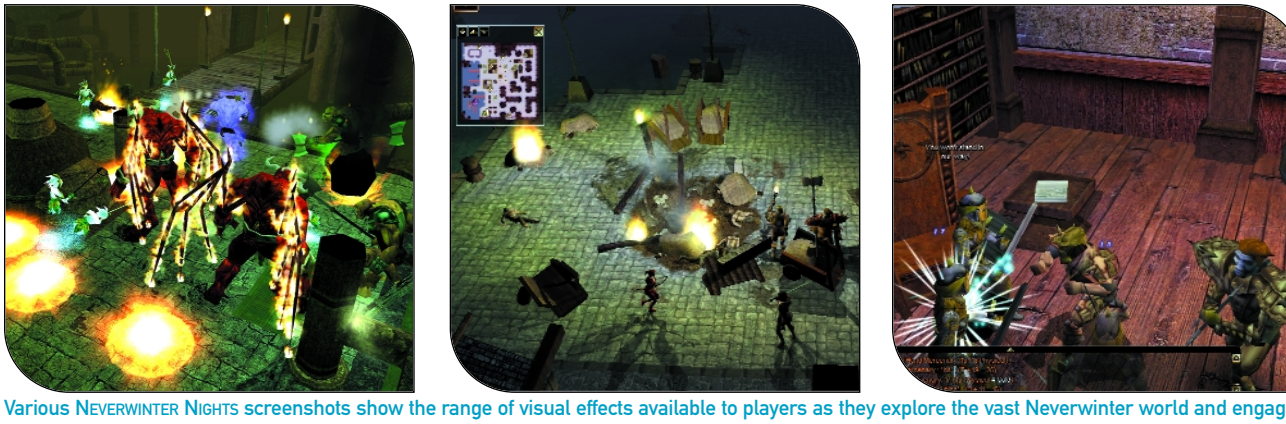
We felt the demos were successful overall and that the incremental PR received from these demos was helpful to the game's market success, but each of these demos consumed considerable team resources. In spite of this impact, the team recognized that demos are a necessary and vital part of the development process — however, they should be part of the schedule and planned accordingly from the start. In our future projects we are booking more time for demos in our schedules, since they always seem to take up more time than originally anticipated.

Everwinter Nights

Though BioWare considers NEVERWINTER NIGHTS a critically and commercially successful product by most generally accepted standards, it is still far from perfect in our eyes. We try hard to learn from our mistakes, and when we run across a hurdle or a challenge we try to avoid getting caught in the details of what happened and focus on the solution. We cast a critical eye on everything from process to user perception. This critical approach often allows us to spot trouble areas ahead of time and plan for solutions before a trouble area becomes a project blockage.

In the end, BioWare is a reflection of the people who work at the company; NEVERWINTER NIGHTS was completed by people devoted to a project they believed in completely; as with many similarly successful products, without their hard work it never would have been possible. But we still have a lot to learn, and we can only try to improve each game in relation to the ones that we released before. Our future games must and will be better still than NEVERWINTER NIGHTS.

Our hope is that the game will help open — and keep open — the door to user-created content for role-playing games. So far things seem to be going well in this regard. As of this writing, there are more than 1,000 user-created modules on the Internet, and this is just a starting point for the hundreds of thousands of players who have purchased the game and who are now using the BioWare Aurora Toolset to make NEVERWINTER modules. We're hopeful that our players will continue to make new content to grow the game's community, and BioWare's Live/Community team will continue to support them in this effort.



Various NEVERWINTER NIGHTS screenshots show the range of visual effects available to players as they explore the vast Neverwinter world and engage in combat.

In Defense of Academe

Illustration by Claudia Newell

Self-taught old-timers like myself, who used to carry each instruction from the RAM to the CPU on our backs, tend to sneer at the idea of academic research in game development. After all, we made it without no highfalutin' degree, and besides none of that stuff's any use in the "real world."

Self-taught old-timers like myself had better watch out, because a new generation of designers and programmers is coming along that will soon show us up for the dinosaurs we are.

Being self-taught has a homespun, Abe Lincoln romance about it, but in real terms it's impractical. If you only learn what you need to know as you go along, you end up not so much self-taught as half-taught, with gaps in your education. I've never had a class in assembly language to this day, and I only know as much about it as I had to learn for various projects. Formal education takes the amorphous mountain of information available on a subject and distills it down into useful knowledge, passing it on efficiently to new generations of students. What would take you 10 or 15 years to learn by yourself through trial-and-error, a good teacher can impart in three or four years of dedicated learning.

It's time that we in the game industry changed our attitude toward formal education and academic research. The academy can do things that industry can't, because it's not constrained by the requirement to build profitable products. It is foolish of us to ignore the opportunity that this represents. Game developers often say that academic research doesn't help them to make better games, so they don't care about it. That's a short-sighted attitude. Other businesses, such as the semiconductor and pharmaceutical industries, have close relationships with universities because they know they will pay off in time.

If academic research isn't helping you to make better games, I can think of three reasons why not:

Game research is just getting started. It's still an uphill battle to do work on games at many universities; senior faculty doesn't respect it and there aren't many sources of funding. Serious work on games has only started in the last two years or so (with certain exceptions in the field of artificial intelligence), but it has begun. If developers disparage it too, we're further hindering the process. If instead we help, then a few years down the line there will be rewards to reap.

You may not be paying close enough attention. Much of the work presented at SIGGRAPH is academic research. There are still too many developers who don't bother to find out what's going on in university research labs.

You may not be pushing the envelope. Academic research is necessarily at the frontier — in fact, beyond the frontier, out in regions where there's no real way to know whether it's

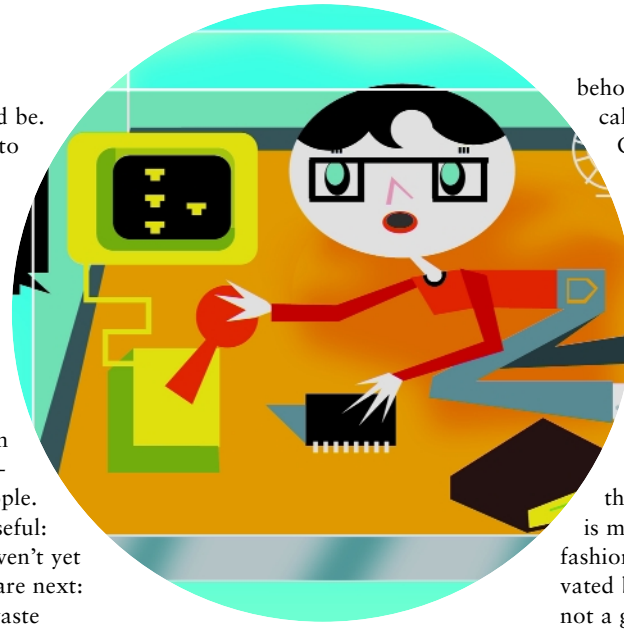
continued on page 55

continued from page 56

useful or not. This is as it should be. Since the academy doesn't have to sell its work, it can look into areas that are really speculative. If you're making a nice, safe game in a nice, safe genre, of course you don't need the academy; you're not taking any risks.

Someone at this year's IGDA Academic Summit at the GDC pointed out that the industry can get four useful things from academe: code, cred, theory, and people. Code is the most immediately useful: software that does things we haven't yet learned to do ourselves. People are next: educated workers so we don't waste time and money training them. And theory comes in the form of advances in all sorts of areas that would be useful to us: AI, simulation, physics, graphics, and so on.

Cred (cultural credibility) is the vaguest, and it has the least immediate use but the most long-term value. Academic study of a medium is a vital stepping stone on the way to its public recognition as an art form. If you have to learn something by yourself, it's probably just a hobby. But if they research it and teach it in college, then it acquires credibility with the public. I'm sure most people thought the idea that movies could be "art" was idiotic in 1915, but they don't now, and the existence of film schools has a lot to do with that.



You may be content for videogames to remain merely popular culture; they're here to stay and the job will keep you fed until you retire. But if you have any aspirations for this medium, if you want to see it achieve some of its extraordinary potential, then it would behoove you to support both theoretical and applied research.

Commercial games can and occasionally do stagnate creatively as long as they make money. It was partly this sort of creative stagnation that led to the crash of 1983. Game publishers were making so much money that they quit bothering to innovate; then they wondered why the public was suddenly sick of videogames.

Competition drives advances by the game industry, but competition is mindless and dependent on the fashions of the moment. It's only motivated by a desire to beat the other guy, not a genuine wish to explore. For serious exploration beyond the frontiers of gaming, we need academic researchers willing to tramp those woods and ford those rivers for the sake of knowledge alone. Let us be Jefferson to their Lewis and Clark.

ERNEST ADAMS | *Ernest is an independent game design consultant, with 13 years' experience in the industry. He also writes the "Designer's Notebook" columns for Gamasutra.com. His professional web site is at www.designersnotebook.com.*