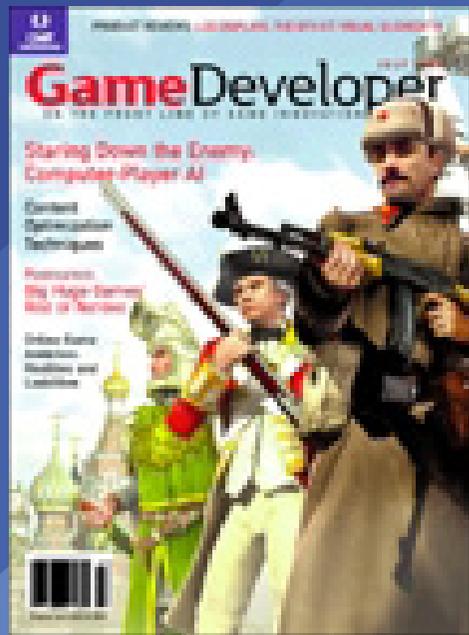




GAME DEVELOPER MAGAZINE

JULY 2003





GAME PLAN

LETTER FROM THE EDITOR

Sequels 2K3: Beyond the Return of the Sequels

There were so many sequels at this year's Electronic Entertainment Expo that next year I fully expect to be attending a show called "E4." However, unlike many, I don't see the prevalence of sequels in a year such as this one as a harbinger of certain death for creativity in games. Rather, a marketplace crowded by product of similar quality inevitably drives innovation for the sake of differentiation.

Sony, lord and master of all it surveys — to the tune of 50 million Playstation 2s, 100 million original Playstations, and well over 1 billion pieces of software sold for both — bats its seductive EYE TOY alongside the next GRAN TURISMO. Microsoft, whose relatively sparse stable of exclusive titles have had difficulty combating the Xbox's reputation as a Portbox, is positioning itself to grab the transmedia brass ring through the carefully concerted push of features such as Xbox Live and new efforts such as the Xbox Music Mixer — the lightbulb of the integrated online living-room entertainment hub may finally go off over the heads of Xbox owners.

Besides these platform innovations, if the majority of the console software on offer looked like rehashes of hardware launch titles (with multiplayer added), that's what console midlife is really about: milking it. A well-sowed console needs its bountiful harvest to help offset R&D costs in the long run. Meanwhile, PC games start to lift the skirt on what the next generation will bring, and graphics are only the beginning. Emergent gameplay through nonscripted interactions is redefining game design the same way that texture mapping redefined graphics.

For developers, the onus is now on polish, both in production values and in gameplay. What dogged determination used to be applied to massaging performance out of hardware must also be applied to animation, effects, audio,

visual design, and gameplay fluidity, whether you're developing for Game Boy Advance or the latest PC graphics hardware. Real polish happens neither overnight nor by accident; it has to be scheduled and budgeted for, which unfortunately leaves smaller, independent developers at a disadvantage unless they are in a position to leverage sufficient resources from a mutually committed publisher.

Polish is what consumers are now able to discern most quickly. Games that have similar graphics aren't a threat to market health as much as games that all feel and play the same. Enticing polish out of a development project doesn't require just money and a comfortable schedule; developers must reexamine their processes to enable refinement to occur at this level throughout the development cycle, from gameplay prototyping to art direction to entity planning to play-testing.

Finally, in the spirit of market differentiation, I'd like to present a list of 30 words that, when used in the title of a game, render me unable to recall the game or anything unique about it, a phenomenon recently tested and proved over the course of three days of E3. The Oxford English Dictionary counts 171,476 words in current use. Next year I'd like to see a few more of them, and fewer of these:

Rogue. Call. Black. Destiny. Elite. Ops. Spear. Hunt. Dungeon. Ninja. Delta. Warrior. Ghost. Strike. Command. Dark. Legend. Star. Knight. Spy. Assault. Evil. Force. Shadow. Siege. King. Underground. Combat. Sword. Pokemon.

Jennifer Olsen
Editor-In-Chief

GameDeveloper

www.gdmag.com
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

Art Director
Audrey Welch awelch@cmp.com

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

Contributing Editors
Jonathan Blow jon@number-none.com
Hayden Duvall haydend@3drealms.com
Noah Falstein noah@theinspiracy.com

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

ADVERTISING SALES

Director of Sales/Associate Publisher
Michele Sweeney msweeney@cmp.com t: 415.947.6217

Senior Account Manager, Eastern Region & Europe
Afton Thatcher athatcher@cmp.com t: 828.350.9392

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

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

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

Account Representative
Aaron Murawski 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



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 Faletta
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



United Business Media

GamaNetwork

SAYS YOU

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



Designing for Other Audiences

Although I agree with Jennifer Olsen's analysis of the challenges women face in the game industry ("What's Good for the Goose . . .," Game Plan, May 2003), I think her conclusions are incorrect. The reasons women and minorities face challenges in the gaming business have very little to do with any kind of discrimination.

It takes a lot less effort to build a game for yourself than to build one for somebody else. Inexperienced young, male, white game developers lack the skills to design a game that would be of interest to any other demographic group. They build what they know. The industry as a whole then produces only games that interest themselves, creating the feedback loop described in the Game Plan column.

The solution is for game production companies to bite the bullet and put forth the extra effort to hire experienced developers that can actually design a game for demographics other than themselves. I recently put my money where my mouth is and, together with a few fellow developers, funded a game for eight-to-12-year-olds. The game was finished under budget and on-schedule, was profitable in just a few months, and nobody had to work overtime. And oh yes, women made up 50 percent of the development team.

Mike Kelleghan
Brainbarf.com (via e-mail)

Mystaken Identification

As one of the creators of the MYST series, I couldn't help noticing Hayden Duvall's use of our work in "The Price of Progress" (Artist's View, May 2003). The article states essentially that it behooves game creators to reuse technology when doing sequels and uses the MYST series to propel this argument. While I agree with this in theory, using Cyan and the MYST series as the poster



Editor-in-chief Jennifer Olsen replies: *The images in question were selected and captioned by the section editor, not by the author, so we take responsibility for both the inaccurate identification and the general muddling of the larger issue. We regret the error.*

Inner Product Columns at Work

I'm getting into a higher-level scripting system for player/entity state variables, so that we can script some time-sensitive performance-based rewards

It takes a lot less effort to build a game for yourself than to build one for somebody else.

child for reuse of technology is pretty off the mark. In fact, the two MYST series images in the article are not only incorrectly described, they actually disprove this theory completely.

The image at the left is supposedly from "the original MYST." The purpose of this image is to show how technology for an older product was used in its sequel. Actually, this shot is from our current project, URU: AGES BEYOND MYST, which uses a completely new proprietary engine we've developed here at Cyan Worlds.

The image at the right is stated to be from MYST III. Kudos — this image *is* actually from MYST III, but unfortunately MYST III used a completely different proprietary engine, and it isn't even ours. Cyan Worlds licensed MYST III to Presto Studios (may they rest in peace).

Contrary to the way the MYST series is portrayed in the context of technology reuse, Cyan Worlds (for better or worse) may be riding the most varied production/technology process in the history of a computer game series.

Joshua Staub
Cyan Worlds (via e-mail)

much like Jonathan Blow's November 2002 column ("Toward Better Scripting, Part 2," The Inner Product) describes.

We've actually been using it for a while to trigger special moves based on stick input. So for example, one move is a reverse backflip that gets triggered if the player is pushing the stick hard in one direction, then quickly pushes it the opposite direction and hits a jump button. With a third-person tethered camera, the axis of direction is arbitrary, since the mapping of stick direction to player movement is camera-relative. By checking the ratio of the covariance major axis length to minor axis length over one of the shorter timeslices ("My Friend, the Covariance Body," The Inner Product, September 2002), we can detect a quick 180-degree stick direction change along an arbitrary stick axis.

Nate Burgess
Digital Eclipse (via e-mail)





INDUSTRY WATCH

KEEPING AN EYE ON THE GAME BIZ | *everard strong*

Falling dollars, rising units. The NPD Group reported that total U.S. game industry sales fell 2.4 percent in the first quarter of 2003 compared to the same time last year. However, unit sales grew 7.4 percent. The decline in total sales and increase in unit sales were attributed to price cuts of hardware systems in the second half of 2002. Videogame hardware experienced a 12.6 percent drop in dollar sales but a 16 percent increase in units sold over Q1 last year. Videogame software saw a 6.5 percent increase in dollar volume and a 1.5 percent increase in unit volume. In the PC game market for the same period, dollar sales fell five percent, and unit sales decreased by 3.5 percent. Analysts predict that high-profile PC games such as DOOM III and HALF-LIFE 2 will turn the market around.

Atari and Firaxis deepen relationship. Atari and Firaxis Games announced a new, long-term development and publishing



Atari has teamed up with Firaxis Games to develop and publish new titles based on classics like Sid Meier's PIRATES!

deal that will bring a remake of Sid Meier's 1987 classic PIRATES! to the PC in 2004. Also included is CIVILIZATION CONQUESTS, an expansion to CIVILIZATION III, scheduled for a holiday release. The deal also involves the transfer of rights to Firaxis for several classic Sid Meier properties for future publication, and extends to first rights of negotiation to Atari for two future unnamed games.

New handhelds revealed. Competition heats up. Following Nokia's February introduction of the N-Gage game deck, Sony has announced a new portable game platform, the PlayStation Portable (PSP), which is scheduled for release in late 2004. Planned features include a backlit 480x272 TFT widescreen LCD with a 16:9 aspect ratio; 3D capability; support for stereo sound and high-quality MPEG4 video; and connectivity to a PC, Playstation 2, or other PSPs via a USB 2.0 port. PSP games will be available on 1.8GB Universal Media Disc (UMD) cartridges.

Joining the competition is Tapwave's elusive Helix. It will include Palm OS PDA functionality; a backlit 480x320 reflective TFT LCD, dual USB 1.1 connectors, wireless multiplayer capability, support for high-quality sound and MPEG4 video, and graphics acceleration. Although no specific games have been announced, Tapwave has revealed partnership agreements with Activision, Digital Eclipse, Atari, and Midway Home. 🎮



THE TOOLBOX

DEVELOPMENT SOFTWARE, HARDWARE, AND OTHER STUFF

IGDA releases free business resources. The International Game Developers Association's Business Committee is making available resources for game developers to improve contracts, streamline game submission processes, and educate developers on game industry best practices. You can download these resources from their web site at www.igda.org/biz.

natFX 2.0 available for Maya. Bionatics has released version 2.0 of their plant-modeling software for Maya users. The version will include the hybrid 2D/3D technology found in their newly-updated version 1.8 for 3DS Max. www.bionatics.com

Kaydara to release new Motionbuilder. Kaydara is planning to release a new version of its 3D character animation soft-

ware, Motionbuilder 5.0, in July. New features will include a storytelling timeline integrating 3D animation with audio, video, and camera shots; improved character animation tools; and customizable shortcuts for better workflow. Motionbuilder 5.0 will be available on Windows 2000, Windows XP, and Mac OS X for \$3,495. www.kaydara.com

WildTangent goes retro. WildTangent revealed their newest game platform, RetroDriver, giving game developers the ability to create and publish 2D online games without needing Java. The platform supports a range of programming languages and includes all of the nongraphical features of Web Driver. The RetroDriver SDK will be free for developers to download. www.wildtangent.com



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

UPCOMING EVENTS CALENDAR

SIGGRAPH 2003

SAN DIEGO CONVENTION CENTER
San Diego, Calif.
July 27-31, 2003
Cost: \$50-\$950
www.siggraph.org/s3003

GAME DEVELOPERS CONFERENCE EUROPE

EARL'S COURT CONFERENCE CENTRE
London, U.K.
August 26-29, 2003
Cost: \$150-\$650 (+VAT)
www.gdc-europe.com

ECTS

EARL'S COURT CONFERENCE CENTRE
London, U.K.
August 26-29, 2003
Cost: Free for industry members
www.ects.com



20-Inch LCD Display Roundup

by spencer lindsay

My employer, Rockstar San Diego, is presently switching over to flat-screen displays for a multitude of reasons:

Flat-screens use much less power, take up less space, are easier on the eyes, and are less blurry. However, flat-screens have their share of disadvantages too: the resolution is not as high as some CRTs, and the color is not as accurate as a tube monitor, though the latest generations are much better at color accuracy.

For this review, I gathered four popular monitor makes and models and hooked up each one up to a 2.3GHz Intel box with an Nvidia GeForce4 Ti 4800 SE, comparing them side by side based on common criteria: screen size (20 inches), native resolution (1600×1200), price range (\$1,000 to \$2,000), availability, and expected usage (game art creation). I ran a basic animation exercise through each, creating first some texture maps in Photoshop, bringing these into Maya, sculpting a basic 3D object, and then running it through some animation sequences. As I tested each step, I looked at how the monitors processed and output the data: How did the colors differ on each monitor? What kind of detail did each offer? Was there a lot of gamma correction needed? What kind of software came with it, and did that affect how Photoshop or Maya interacted with the screen? How was customer support? Were the on-screen menus easy to use?

Though a couple of the displays had more bells and whistles than others, overall they were all a much better choice than a CRT.

Planar PL201M

We have a few of the 19-inch CT1904M Planar monitors here at work, but the 20-inch for some reason seems huge in comparison. The PL201M has more resolution, more screen space, and a better swivel mechanism. Setup was a snap, and the color quality was excellent once I figured out the menus.

The Good. I pulled the monitor out of the box, plugged it in, and off I went. Once I figured out the confusing menu system, there was little need for fine-tuning. The integrated speakers and headphone jack on this model were a nice touch. As with all the monitors I reviewed, the monitor angle was adjustable to a point; not as moveable as the new Apple iMac, but very adaptable. As with every monitor I tested, there was a slight divergence of colors in the blue/magenta spectrum of the Photoshop color picker, but other than that it performed well in all my tests. The thin form factor left plenty of depth on my desktop once it was installed.

The Bad. Color tuning was a bit difficult due to a confusing menu, and the bezel could be a bit thicker to keep the background off of the screen (what's behind the monitor clashes with the data on-screen).

NEC LCD2080UX

On first glance, the ultra-thin bezel on this monitor looked really cool, like a flat sheet of graphics perched on my desk. But, after using it for a while, the absence of a bezel became a detriment, as whatever was behind the monitor (say, a *Star Wars* poster) mixed in with the data on-screen. A frame around the screen helps the eyes not to focus on the background when looking at the edge of the monitor.

The Good. The NEC installed flawlessly with no need for manufacturer's drivers, and color tuning was all set up; I didn't have to tweak it at all to get rich colors and good contrast. When I did fiddle with the menu system, it was easy to understand and navigate. Testing showed this monitor to be a great choice for developers who have big black walls behind their desks.

The Bad. The bezel on this monitor is too thin. This might be a personal taste issue, but I found that it didn't keep the distracting background of my messy desk from interfering with the screen. Also, despite wearing the highest price tag of all four monitors tested (\$1,699), this monitor didn't have the bells and whistles I found so appealing in some of the other monitors.

Dell 2000FP

For console developers, a good space-saving feature of a flat-screen monitor is the ability to run component video through it. Getting that big TV off your desk is a definite plus. The only disadvantage of running the console

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



NEC LCD2080UX



Planar PL201M



Dell 2000FP



Sony SDM-X202

through your main monitor is if you have PC-based tuning widgets that you need to access while viewing the game. Although the Dell was the only monitor that had the component video input feature, this feature is available from Sony at a higher price range. The display was a bit blurry until I installed the correct drivers supplied with the monitor, but once I got the thing set up, it performed well through all the tests.

The Good. Another easy install and a very clear display once the supplied drivers were installed. The S/component video input is a great addition for console developers.

The Bad. Setup was a bit more involved and required special drivers.

Sony SDM-X202

The Sony SDM-X202 has all the bells and whistles and is priced better than many comparable flat-screen monitors. Setup was a breeze, the display clarity and color were amazing, and the form factor was nice and compact. Although I had some troubles getting the Wacom tablet to come up at first, I updated the Wacom drivers and the problem went away. USB, dual audio I/O, and a set of speakers take care of a lot of extra desk-top gizmos you would need with a less gadgety monitor. Having two USB input channels allows you to use this monitor as a KVM switch if you use a digital and an analog input.

The Good. The monitor was easy to set up and performed well through all the tests. The speakers and dual audio I/O combined with the switchable USB connections makes this monitor perfect for game development, especially for those of us with two computers. In addition to

the bells and whistles, the unit is well-designed, neatly hiding the cables and connectors behind a shield.

The Bad. There is only one extra gizmo I'd like on this monitor, and that's a component video input. Although S/component video input is an option on other monitors, it would make this unit the perfect game development screen. I did have some startup problems with my Wacom tablet, but those were resolved by updating from the Wacom site.

Although all of these monitors were excellent in different ways, the Sony was definitely the superior unit, with its multitude of extra components. Based on all my tests I'd first recommend the Sony SDM-X202, while a good second alternative would be the Dell 2000FP.

Tommy Tallarico Studios' SFX Kit

by aaron marks

Sound Ideas has released a new sound effects library designed specifically for the game and interactive entertainment industries, and it's pretty darned good. The SFX Kit, created by Tommy Tallarico Studios, boasts seven CDs comprising 19,655 game-relevant .WAV sound effects, arranged in 110 categories with an included searchable database for both Macintosh and PC. This collection's highlight is its utility — practically every sound effect can be plugged straight into a game off the shelf, either as a one-shot or ready-made loop. The library even comes packaged in a nice leather pouch for convenient portability.

Sound categories are far-reaching and grouped logically on each disk, ranging from aircraft, vehicles, and machinery to human sounds and footsteps, monsters and magic, sports, weapons, and sounds of destruction. Plenty of original sounds make production in any game genre top-notch. The eighth CD is the data disk, containing files in Excel format, comma-separated text, tab-delimited text, and a fully functional, searchable version of FileMaker Pro 5. With these, you can import the sound lists into current search software, or easily import any previous sound effects database into FileMaker.

For years, game sound designers who use the many film sound libraries available on the market have had to do so with a lot of extra work. Most libraries are released as audio CDs, and for game guys, ripping files between audio and .WAV formats can be cumbersome. With The SFX Kit, you can examine individual sounds right from the editor, and open it directly or save it to

MONITOR ROUNDUP

All 20-inch monitors reviewed had an on-screen resolution of 1600x1200, with 16 million viewable colors.

- PLANAR PL201M ★★★★★
www.planar.com
(503) 748-1100
STREET PRICE: \$999
- NEC LCD2080UX ★★★★★
www.necmitsubishi.com
(888) 632-6487
MSRP: \$1,699
- DELL 2000FP ★★★★★
www.dell.com
(800) 999-3355
MSRP: \$999
- SONY SDM-X202 ★★★★★
www.sonystyle.com
(877) 865-7669
MSRP: \$1,499

your hard drive. It's quick, painless, and saves a tremendous amount of time with a large project. This feature is by far the biggest selling point for me, so my hat's off to Tommy and his crew for making game audio professionals' lives easier.

But what good is a sound library unless it sounds good? The SFX Kit is professional grade both in quality of the sounds and the creativity behind them. I was able to listen to almost every sound in the library while working on a recent project, and except for a few overmodulated files and a limited amount of stereo files, the 44.1 kHz, 16-bit files sounded excellent. Joey Kuras, the creator of most of this library, is known in the industry for his sound design talent, and it is a treat to have his creations available for use.

Any discriminating sound designer knows you are only as good as your tools. With this set, you'll sound professional whether the sounds are used out-of-the-

box or for use in making fresh, innovative sounds. At \$695, this set is great for anyone in the videogame business who is serious about making their sounds stand out.

★★★★ | **The SFX Kit** |

Tommy Tallarico Studios |

www.sound-ideas.com/sfx-kit.htm

Aaron Marks is the author of The Complete Guide to Game Audio (CMP Books).

Visual SlickEdit 8

by *justin lloyd*

What has always been likable about SlickEdit is its start-up speed. It's as fast as Notepad, making it an ideal replacement general text editor. It opens Microsoft Developer Studio projects faster than Dev Studio.

SlickEdit supports a wide range of lan-

guages and project environments, and there's new support for pre- and post-build steps. It now supports Borland JBuilder, Visual Studio .NET, C# for Linux and UNIX, several scripting languages (Lex, YACC, ANTLR), Verilog, SAS, and XML schemas, and has improved support for Apache Jakarta Ant and various flavors of makefiles. XML editing is easy and powerful using a tree view, and SlickEdit lets you add, remove, and search XML elements and XML attributes via either local or HTTP-located DTD.

SlickEdit's tagging feature automatically tags source files in a project, looking for keywords, function definitions, and class declarations in each file, and builds a navigation map. The map enables the auto-completion features (such as parameter type matching, class/structure member lists, and syntax expansion) to operate, and aids in moving around projects.

PROFILES

TALKING TO PEOPLE WHO MAKE A DIFFERENCE | *everard strong*

Step 1: Make Games Step 2: Don't Suck

Brian Fiete Distills PopCap's Browser-Based Success

With broadband capabilities finally reaching an oft-trumpeted mass-market that failed to materialize in the early days of 2000, browser-based games are experiencing not so much a resurgence as outright commercial success. PopCap, founded in 2000 by friends Brian Fiete, John Vehey, and Jason Kapalka, has seen its success — backed by such popular titles as BEJWELED, BOOKWORM, and INSANIQUARIUM — rise alongside broadband-based entertainment's rising popularity.

Game Developer talked to Brian Fiete about how PopCap survived those earlier, unstable times and how they are transitioning to stay fresh and grow — both as a company and in their product selections — in an expanding industry.

Game Developer: When you formed PopCap in 2000, what promise did broadband hold for you?

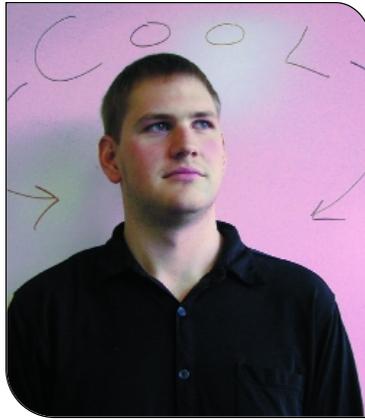
Brian Fiete: In the beginning we were attracted to creating web games because of the huge potential audience, the shorter development cycles, and because most of the web game content that existed around that time was terrible. We really felt that if we created kick-ass games that people loved there was no way we could fail. Developing web games was never supposed to be a terrifically clever business move, we were only interested in staying independent and making enough money to afford ramen, rent, and spiffy computer upgrades.

GD: You started offering enhanced downloadable versions of your games. How and why did you come up with that strategy?

BF: Things were looking a bit bleak before we tried selling Deluxe versions of our games. People were a bit pessimistic about that concept, though, as shareware titles rarely bring in any real money for anyone. Now that the Deluxe versions are selling like hotcakes, we pretend we knew it would work all along.

GD: Can you give us an idea of what your sales have been like since PopCap's 2000 launch?

BF: 2001: a little money, 2002: millions, 2003: junkloads. The long version: We were just starting out in 2001 and though we had what we thought were a lot of sales and were generating \$30-40,000 a month, it was nothing compared to what we did in 2002 once we had more products, understood the business, and had partnered with most of our target major portals. 2003 is turning out to be our best year yet, since now everyone under-



"The last name is pronounced 'fiet,' as in stinky feet, big feet. 'Mr. fatty fatty feet man.' Now you can understand why I was shunned and turned to the dark, dark world of game development."

stands marketing the products better, we have more hits, and we're making more money and sales from markets such as handhelds and mobile phones.

GD: Forming a company with three people, did you have a clear division of labor before forming PopCap? How has that changed with growth?

BF: Jason did the game design and art, I did all the programming, and John was responsible for handling the biz stuff. The team dynamic was great, things instantly clicked, and we made a lot of progress in a short amount of time. Now that we've expanded I think we've done a good job building out a team that has the same spirit we started with.

GD: Did you start PopCap with a detailed mission plan of where it was headed?

BF: I think the mission plan was something like "Step 1: Make games. Step 2: Don't suck." But aside from a desire to do the right thing in web game development, we really

didn't have any idea. Since launching, we have found that we can make compelling games by distilling a concept down into a small but complete and consistent experience. These days we're more likely to be looking for ways to trim a product down even more than to add to it.

GD: In developing new games for browser-based use, what are your top considerations?

BF: The most important thing is that it must be instantly understandable. If a user feels confused during even the initial presentation of the game, it's likely he will move on to some other game. Even if the game had all sorts of complexities under the surface, you must present it in a way that is very approachable where new game play concepts are slowly added. Aside from that, size and compatibility are important (we try to make sure our games work on 1998-era computers with 4.0 browsers and 56K modems). Creating an interface/control scheme that works well in a browser window is important too. Oh, and stay away from arrow-key controls.

GD: How do you instill a creative, while still productive, atmosphere at PopCap?

BF: For me, creativity is the spark which fuels production. A great idea has little value until you see it through to implementation. At PopCap we feed off of each others ideas and we all share in the high of building something that we can really be proud of. I guess the big thing is that at the core we are all just naturally hard workers and live for the challenge of figuring out how to make the next game the best one we have ever made. 🍌

Unified Rendering LOD **Part 5**

This series of articles is about level-of-detail for large triangle soup environments. Such environments must be divided into chunks; the biggest issue in this kind of system is filling in the holes between neighboring chunks at different levels of detail. I fill these holes with small precomputed triangle lists called “seams,” and earlier in this series I showed how to construct these seams.

In Part 3 (May 2003), I showed how to split a single chunk of geometry into two, and how to generate the initial seam between them. Last month, in Part 4, I split a chunk that is connected via a seam to another chunk, while maintaining existing manifolds. I mentioned that this is sufficient to build a chunked BSP tree of the input geometry, and I showed some code that split the Stanford Bunny by arbitrary planes and played with the LOD of the various pieces.

This month, I’ll begin with that basic premise of building a chunked BSP tree. Before long I’ll be rendering a large environment with this system, not just a bunny.

To render a large environment I’ll focus on three major parts of the algorithm: (1) dividing the environment into initial chunks, (2) putting those chunks together into a detail-reduced hierarchy, (3) choosing which levels of the hierarchy to display at run time.

Dividing the Environment

I don’t want to assume anything about the shape of the environment. Maybe it’s a big localized block like an office building, or maybe it’s a loose set of tunnels that curve crazily through all three dimensions. An axis-aligned approach to segmenting the geometry, like a grid or

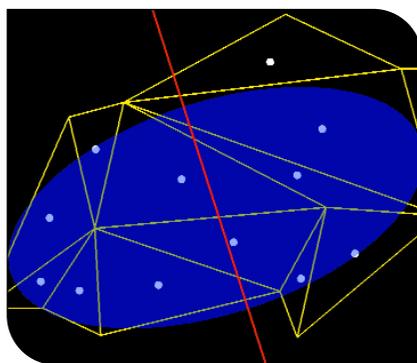


FIGURE 1. Some triangles (yellow); their barycenters (white dots); the ellipsoid, drawn at 1.7 standard deviations, that we get from treating these barycenters as point masses (blue); and a splitting plane that spans the minor axes and passes through the center of the ellipsoid (red).

octree, would work well for the office building but not for the tunnel. The chunked BSP treatment is nice because it doesn’t much care how pieces of the environment are oriented.

I needed a method of computing BSP splitting planes. Most game-related BSP algorithms divide meshes at a one-polygon granularity, so to acquire a splitting plane they pick a polygon from the mesh and compute its plane equation. But that’s the kind of thing I am trying to get away from — thinking too much about single triangles violates the spirit of the algorithm. I want a more bulk-oriented approach, so I statistically compute a plane that divides the input into two subchunks of approximately equal triangle counts.

There are other things I might wish to do besides minimizing the difference in triangle count between the two sub-

chunks. I might wish to produce subchunks that are as round as possible, so that I don’t end up with a scene full of long, sliver-shaped chunks. I might wish to balance the number of materials used in each chunk. Most probably, I’d whip up a heuristic that combines many parameters and makes trade-offs to achieve a happy medium.

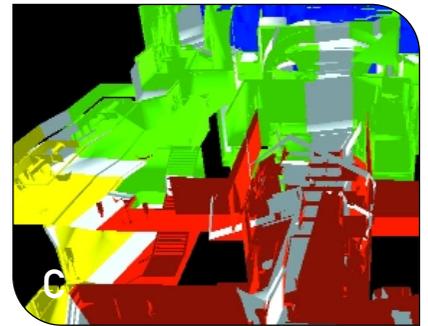
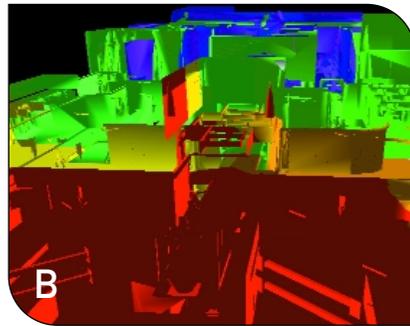
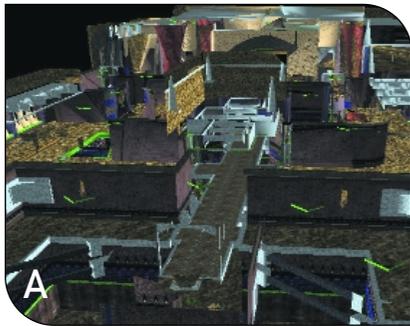
To compute the splitting plane in a quick-and-dirty way, I treat the chunk as a bunch of point masses, one mass located at the barycenter of each triangle. I then compute the vector-valued mean and variance of these masses (see “My Friend, the Covariance Body,” *The Inner Product*, September 2002). This gives me an ellipsoid that summarizes the mass of the chunk, as though that mass were normally distributed. I then split that ellipsoid in half, using a splitting plane that passes through its center and spans the ellipsoid’s two minor axes. I use the minor axes because this produces halves of minimum eccentricity — in other words, pieces that are “the most round.” (I might produce pieces that are more nicely shaped by choosing not to split the ellipsoid through the center). See Figure 1 for a 2D illustration of the splitting-plane computation.

Combining Chunks into a Hierarchy

This LOD rendering algorithm doesn’t place many constraints on the way I organize my chunks. It wants an n -ary tree with hierarchical bounding volumes that help me choose detail levels. I’ll talk about computing those volumes in the



JONATHAN BLOW | *Jonathan can be reached at jon@number-none.com. There’s enough built-in stress in marriage without noise contributing.*



FIGURES 2A-C. (A) QUAKE 3 level “hal9000_b_ta.bsp,” by John “HAL9000” Schuch, rendered using the LOD system. (B) Chunks have been drawn with color coding showing their resolution level. (C) Level chunks artificially moved apart so you can see the seams (white bands).

following section; for now I’ll only discuss putting the chunks together.

First, I need to decide on a policy about which chunks to combine. The easiest choice is to use the hierarchy that was created by the BSP splits. I would combine two chunks into one, detail-reducing to a given triangle budget at each level, until I once again reach the root of the BSP tree.

This is easy because I don’t have to think much, and in fact it’s the first thing I implemented. Clearly, though, it may not produce the best results. For one thing, in any tree structure like this I end up with nodes that are neighbors spatially but far apart in the hierarchy. In terms of scene quality it may be best to combine these nodes early, but they won’t be combined until the level of detail becomes very low. Second, I’d like the freedom to combine more than two chunks at once. The preordained BSP hierarchy does not provide much freedom.

So instead, I take the chunk geometry stored on the leaves of the BSP tree, throwing away the tree structure itself. Then I cluster the chunks using a heuristic that is independent of the original chunking process. This lets me match any number of chunks together, building a new n -ary tree from the leaves to the root. At each level of the n -ary tree, I combine the participating chunks and seams into one mesh, and then run a static LOD routine on that mesh. This is the same thing I did with the terrain hierarchy from Parts 1 and 2 (March and April 2003), but now the environment representation is less restricted.

You can imagine many different policies for choosing the triangle budget for each node in this geometric hierarchy. In the sample code, my heuristic is that each node should have approximately the same number of triangles (for example, 4,000). So the BSP splits occur until each leaf has about 4,000 triangles; then I discard the BSP tree and begin combining leaves four at a time, LOD-ing each combination back down to 4,000 triangles. So in the end, I have a 4,000-triangle mesh at the root that represents the entire world, with subtrees containing higher-resolution data.

Choosing Which Resolution to Display at Run Time

So these hierarchical chunks are representing world geometry; at run time I need to choose which chunks to display. Typically, LOD systems estimate the amount of world-space error that would be incurred by using a particular level of detail; they project this error onto the screen to estimate how many pixels large it is, then use that size in pixels to accept or reject the approximation.

Algorithms from the continuous-LOD family tend to perform this operation for each triangle in the rendered scene. However, I want to operate at a larger granularity than that. I could do that as follows: for each chunk, find the longest distance by which the LOD’d surface displaces from the original, pretend this error happens at the point in the chunk

that is nearest to the camera, and use the corresponding Z-distance to compute the error in pixels.

It’s customary in graphics research to take an approach like this, since it’s conservative: it’s possible to achieve a quality guarantee, for example, that no part of the chosen approximation is rendered more than one pixel away from its ideal screen position.

But with graphics LOD, this kind of conservatism is not as fruitful as one might assume. When choosing LODs based on the distance by which geometry pops, such a system ignores warping of texture parameterizations, changes in tangent frame direction, and transitions between shaders. The popping distance can be vaguely related to some of these things, but in general it only controls a portion of the rendering artifacts caused by LOD transition. Even with a strict bound on popping distance, there is no guarantee about the quality of the rendered result.

In addition, this conservative approach can be harmful to performance. To provide that somewhat illusory guarantee, the algorithm must behave quite pessimistically (for example, assuming that the maximum popping error occurs at the position in the chunk closest to the camera at an angle that is maximally visible). This causes the algorithm to transition LODs earlier than it should need to.

I suggest we acknowledge the fact that, when tuning LOD in games, it’s standard practice to tweak some quality

parameters until we find a sweet spot that gives us a good balance between running speed and visual quality. So it doesn't matter how many pixels of error a particular LOD parameter represents, because we don't set the parameter by saying, "I want no more than 0.7 pixels of error," we set it by tweaking the pixel-error threshold until we decide that the result looks good. Whatever number of pixels is produced by that tweaking, that's what the answer is. Conservative bounds are not specially meaningful in this situation.

In this month's sample code (available for download at www.gdmag.com), I approximate the spatial error for a given LOD without using a conservative worst-case. I do this by finding the root-mean-square of all the spatial errors incurred during detail reduction. Recall that I'm using error quadric simplification, so squared spatial errors are easy

to query for. I take each quadric Q and zero out the dimensions having to do with texture coordinates, which gives me Q' . Then I find $v^T Q' v$, where v is the XYZ-position of a vertex in the reduced mesh and Q is its error quadric. That product yields the squared error; I find the mean of all those squared errors and square-root that mean.

This gives me an error distance in world-space units. I use this distance to compute a bounding sphere centered on the chunk, whose radius is proportional to the error distance. If the viewpoint is outside the sphere, I am far enough from the chunk that I don't need more detail, so the chunk is drawn. If the viewpoint is inside the sphere, I need more detail, so I descend to that chunk's children.

The radius is proportional to the error distance because perspective involves dividing by view-space z . For chunks within the field of view, view-

space z is approximately equal to the world-space distance from the camera to the chunk, which is why world-space bounding spheres make sense. The constant of proportionality is the LOD quality level, which you can adjust at will. You need to make sure that the spheres are fully nested in the chunk hierarchy, which means potentially expanding each node's sphere to enclose its children's spheres.

These spheres can be viewed as isosurfaces of continuous-LOD-style pixel error projection functions. For more in the way of explanation, see my slides from Siggraph 2000, as well as Peter Lindstrom's paper (both are in the For More Information section). Both references describe continuous LOD schemes, so I don't recommend you implement either, but the discussion of isosurfaces can be useful.

Back to talking about nested spheres.

To keep things simple, I have so far pretended that an LOD transition is performed exactly when the viewpoint crosses one of those spheres. But as you know from Part 2 (April 2003), we actually want to fade slowly between LODs. In the sample code this is done by computing two radii, one that is greater than the instant-transition radius, and one that is less than it. LOD fading begins when the viewpoint crosses one sphere and ends when it crosses the other.

You can easily imagine more sophisticated methods of computing LOD sphere radii. Perhaps you want to render approximately the same number of triangles every frame, no matter where the player is standing in the environment. You could run an energy minimization system at preprocess time, expanding and contracting the spheres until you achieve something close to the desired result. Or instead of tweaking

radii, you could leave radii fixed and adjust the actual number of triangles used in each LOD chunk.

Sample Code and Future Work

This month's sample code preprocesses and renders QUAKE 3 .BSP files. You can see the results for one such environment in Figures 2a–c. The code does everything discussed in this article. It's not yet a plug-and-play system for generalized use, but it's a good starting point if you want to do LOD experiments or make custom tools.

Because the system stores triangle soup meshes, the file size for the environment might end up being very large. A lot of space is needed to store a mesh of a height-field terrain than for height samples. To alleviate this problem, it may be worthwhile to investigate mesh compres-

sion. Perhaps that will be a good topic for some future articles. 🐉

FOR MORE INFORMATION

- Blow, Jonathan. "Terrain Rendering Research for Games." Slides from Siggraph 2000. Available at www.red3d.com/siggraph/2000/course39/S2000Course39SlidesBlow.pdf
- Lindstrom, P., and Pascucci, V. "Visualization of Large Terrains Made Easy." Proceedings of IEEE Visualization 2001. Available at www.cc.gatech.edu/gvu/people/peter.lindstrom
- Gottschalk, S., M. C. Lin, and D. Manocha. "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection." Proceedings of Siggraph 1996. Available at <http://citeseer.nj.nec.com/gottschalk96obbtrees.html>

Genre Art, Part 1: Working within an FPS

As anyone who is familiar with my writing will be aware, I don't know too many long words. If I manage to slip in the occasional "serendipity" or "falsification," I am happy. Today, however, I have finally found myself in a position to use a word of no fewer than 20 letters in length — if only my high school English teacher could see me now. And what is this word, you ask? The word is "compartmentalization."

Now, I know that for a word of this length it doesn't have the intellectual overtones of something like anthropoginian, or rhombicosidodecahedron, but unlike those words, compartmentalization affects all of us in one way or another throughout our lives, and it has some bearing on game art. If you look at my photo that accompanies this article, you will notice two things: first, I am about as photogenic as a bag of potatoes, and second, my hair is not of traditional coloration. Not surprisingly, blue (or whatever color) hair is the perfect excuse for complete strangers to place me in a box labeled "Warning: Social Deviant. Keep away from children and small animals." It doesn't matter that I am a father of four who drives a minivan and likes cooking, the compartment that I fit into for those who don't know me is the same one that would be chosen for most anyone one with odd hair, clothes, piercings, or tattoos.

Breaking the world down into categories that can be used quickly to interpret what we see and experience, instead of having to evaluate everything fully from scratch, helps people expedite the vastly complex processes of daily life. Making comparisons in our mind to past



Doom III's release is expected to motivate games to upgrade their existing hardware.

experiences and their subsequent outcomes as well as consulting information we have absorbed from our environment helps us to make decisions and gives us a wealth of information that we can draw upon at any time.

The process of compartmentalization is largely implicit or unconscious, but in some areas, grouping things by expectation or generalized content is explicit, specifically when we talk about things such as genre. As with film, videogames are readily broken down into categories where the content is broadly definable as sci-fi, horror, adventure, and so on. There are, however, more specific game types that allow us to further compartmentalize individual games with labels such as FPS, RTS, and sport sim.

Genre conventions are by their nature generalizations and can be viewed nega-

tively if they are seen as stifling originality. On the other hand, it is realistic to assume that now, as with the movie industry, every release will be categorized, with a vague genre assigned whether it fits or not. This tendency is largely to facilitate the marketing, but it's also a by-product of the human urge toward compartmentalization.

In light of this trend, artists in the game industry must face an array of issues that stem from the inevitability of genre art if we are to get the best results from our work. This article will look at the first-person shooter with a view to establishing some of the ways in which we can make the most of the genre rather than simply being overrun by it.

Shooting in the First Person

The first-person shooter has come a long way since the early days of WOLFENSTEIN 3D and DOOM, and perhaps more than any other gaming genre, it places a huge premium on graphics. To some extent you can say this about most types of games at present, but the FPS scene is driven by influential developers fed by the technology race to constantly deliver the next big thing in visuals. In all the graphical advancements over the last decade, many of them have been heralded by a flagship FPS game that exploits the new technology on show.



HAYDEN DUVAL | Hayden started work in 1987, creating air-brushed 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 with his wife, Leah, and their four children, in Garland, Texas, where he works as an artist at 3D Realms. Contact Hayden at haydend@3Drealms.com.

Predominantly a PC-led genre (although this is certainly changing), FPSes attract prominent developers who have traditionally been able to focus on cutting-edge technology that will allow them to aim at high-end systems, regardless of the restrictions that this puts on their potential market. The gaming hard-core are pretty much guaranteed as consumers, and the buzz generated by a “spectacular new game that pushes graphics to the next level” can motivate a reasonable percentage of more casual gamers to upgrade their hardware.

In addition to the graphics focus at the consumer end, a selection of FPS developers are also highly focused on licensing their technology. The game itself then becomes part demo and as such needs to impress with its bells and whistles. As an artist in this kind of situation, it is a great opportunity to sail into uncharted graphics territory, giving us more tools in our bag of tricks but also pushing us to learn new techniques. Outside of this relatively small group, however, there are still many, many FPSes in production using older (and cheaper) licensed technology or internally developed code, and from an artist’s point of view, similar areas that need to be considered regardless of the technology involved.

Weapons

What is the essence of the visual experience of playing an FPS? It’s a hard question to answer, as UNREAL isn’t the same as HALF-LIFE and HALO is hard to equate with something like DEUS EX, but there are some component parts that are similar.

First, an FPS focuses by definition on the shooting. The generalized gameplay premise of most FPS games is to traverse the game world achieving certain goals that advance you toward the end, killing hordes of rampaging aliens, monsters, and/or Nazis as you go. Thus, a player’s weapons are a significant feature of the FPS.

As first-person perspective dictates that the game be played from the point

of view of the protagonist, the player character is either a blank canvas onto which players are expected to project themselves, or as in some cases, the player takes control of an actual character whom they generally only see in cutscenes. As such, the weapon at the bottom of the screen is the player’s main representation of themselves in the game, adding extra significance to this element.

Once weapon types and function are established, the visual design stage is set. Weapon design can be divided into the two categories of real-world and imaginary weapons.

Reality Fights

Any FPS that is set in the real world, whether it is a historic World War II environment or a contemporary city, calls for weapons that are both realistic and distinctive. There is nothing quite so dull as 10 handguns that all appear the same. While many FPS boxes boast statements like “More than 60 unique weapons,” what this often means is “More than 60 variations of the same four weapon types.” So how does an artist go about making real-world weapons look distinctive?

The rather flaccid answer has to be that, in reality, many guns look very similar, so the weapon designer should be mindful of this rather than relying on the player to be able to distinguish between a Ruger and a Glock. A perfectly faithful reproduction of real weapons may excite the gun enthusiasts that buy your game, but differentiating between weapons is generally helped along by a process similar to that of caricature, where prominent features are emphasized to produce a more obvious visual distinction.

In addition to modeling the visual design, variations in finish (matte black, nickel plated) can create distinctions. Sounds, changes in muzzle flash, and unique recoil animations can also help differentiate individual weapons in subtle ways.

The bottom line for a player in this situation is whether or not they feel like

each different weapon is in fact different. Gameplay elements such as accuracy, power, and reload times add a lot to this feeling; artists can reinforce it with visuals that depart from the strict replication formula and go for a method that evaluates design in terms of player satisfaction.

Malice in Wonderland

Imaginary settings (and with FPSes this usually means sci-fi), open up a far wider spectrum of weapon design possibilities. Here the artist’s challenge is focusing on visualizing the weaponry that goes along with the crazy types of weapon functionality designed for the game. What does a Nonlinear Ion Impaction Rifle actually look like, and how can we make it appear distinctive from the Gamma-Field Induction Cannon? The visual design of such exotic fire-power can increase the quality of the player’s experience with each weapon, especially that feeling of excitement the player gets the first time he or she picks up something that looks like it packs enough punch to sink an aircraft carrier.

As a starting point, there is inspiration of all sorts to be had in the world of film, some of it more suitable than others for videogames. *Men in Black* sported some attractive chrome pieces that combined the feeling of highly advanced engineering with a coherent design aesthetic. *Aliens*, on the other hand, brought us the natural evolution of the gritty manmade weaponry we associate with ground infantry, working from weapons we have at present rather than straying too far from what we know.

More recently, nonrealistic FPSes have also aimed to incorporate information that gives the player visual feedback on some aspect of the weapon’s present state. Such feedback might range from a simple ammo counter incorporated into the weapon, to indications of overheating or current ammo type selected. Making each weapon dynamic in some way helps bring them to life and is undoubtedly better than making players divert their attention to a separate area on the HUD.

As far as actual modeling of these weapons goes, the simple exercise of making something look good as it rotates in the level, ready to be picked up isn't the whole story. The player will for the most part see each weapon protruding into the game world at the bottom of the screen at a very specific angle. This is the primary point of view for the player and each weapon must be built with this in mind. While it's O.K. to have the weapon that is carried be slightly different from the one seen in full in the level, any such change must be limited. Putting in some really great detail that is obscured once the player is carrying the weapon is wasted energy.

Having taken care of the player and the weapons, you may notice that there is still a lot of screen space to fill. What do we expect, or perhaps more importantly, what do we want from our FPS game world?

A Love-Crate Relationship

First, let's talk about crates. Crates inspire a love/hate kind of thing where half of you love the utility of a simple wooden box which can be smashed or jumped on and which can hold any number of useful items, and the other half of you feel cheap and dirty for not coming up with a better solution.

As with weapons, the bottom line is that the player wants to have fun. Placing objects in the world that players can interact with in interesting ways is a big plus, but do these have to be crates? Probably not. Is it necessary to tie yourself in knots trying to come up with viable alternatives? I doubt it, but in this case, you may consider putting everyone on "crate duty" for a limited period. Despite how it sounds, crate duty is not a cruel form of punishment where offending artists are locked into a room and forced to put things in boxes until they repent of their evil ways. Rather it's a forum where all artists get to explore their creativity and inventiveness for a restricted time within a specific set of guidelines.



Making weapons look distinctive was a challenge for BATTLEFIELD 1942's artists.

Artists on crate duty are given a certain amount of time to design and create as many viable alternatives for the humble crate as they can, within the context and design limitations of the project. Special prizes are then given for the most innovative or amusing one that can actually be used in the game. Crate duty can be assigned as an area specific task, with the "crate" object customized for individual locations, or in a more generic fashion for game-wide implementation. It may seem frivolous, but this process is a form of extended brainstorming and it can allow artists to bring variety and interest to objects within a game world that are necessary, but that don't necessarily have to be dull.

Environmental Block

Once we are past the crates however, what then? FPS worlds are now much more varied than the cuboid corridors and passageways of years gone by, but certain restrictions tied to engine and platform still place limitations on what can make it to the screen for now. Part of the appeal of an FPS is the intimacy of the first-person viewpoint, where enemies come right at you and your screen can be filled with creatures closing in on you as you frantically search for a weapon with sufficient ammo. Expansive areas are great to add scale to the game, but in many cases, it can be counterproductive to have too much unpopulated space if the pacing is allowed to slip for too long.

Even when exploration is needed, the

FPS genre thrives on action. Building a world that helps focus the player on the action heightens the experience. Artists can create tension for players by fashioning an atmosphere of foreboding. For example, UNREAL 2's many tightly designed sections (specifically corridors) present the player with obvious obstacles that block the player's line of sight and cry out "ambush." Building an area where players feel threatened from both the front and the back also increases tension; players know they must progress forward, while also needing to keep looking behind them.

Lighting variations can darken areas that might contain enemies. As new per-pixel lighting technology presents a sharp increase in what can be achieved with real-time shadow effects (we've all seen how DOOM 3 relies heavily on monsters springing from the darkness), there will be greater opportunity to design levels that use lighting more effectively. Even a simple lighting shut-off is a useful visual mechanism to add tension, where the player is plunged into darkness, hears ominous sounds, and then is immediately confronted with an enemy of some kind when the emergency lighting kicks in (UNREAL 2 uses similar techniques).

Know Your Conventions

There is a lot artists can do with genre conventions that may at first seem like limitations. Artists can play with conventions by using them to their advantage when they want to inform the player without too much exposition, or by turning them against the player to defy their expectations and create greater emotional impact. All you need to remember is that that power lies with you, and using it wisely can have a major impact on the final product.

Next month, I will be looking at the different set of challenges and opportunities facing an artist working in the third-person genre, where character is a major factor and where movement through an environment is central to gameplay. I'll examine how these aspects of the third-person genre affect both the visual design and its execution. 

Getting a Sound Education

“How did you get into this business?”

That is one of the most common questions asked of anyone in game production. The second most-asked question would be the follow-up: “How do I get into this business?” These two questions were addressed at the recent Academic Summit held by the International Game Developers Association. We are now at a point where the first generation of self-taught developers are dwindling, and the next generation is coming from schools where game development is an emerging discipline.

In major schools and universities, departments are forming and merging to address the growing demand for knowledge and experience in interactive media, design technology, digital editing, and (you guessed it) game development. This is not to say that audio and music are not part of the picture, though as a component they are often the last to be thought of. Because of the precedence in audio engineering and music departments, one would think this would give our discipline a head start. But depending on the school and the program, getting it integrated into the other game production disciplines is another story.

Finding that right program or school to fit one’s schedule and/or budget would be the first hurdle. On the side of higher education, schools such as MIT and CalArts take the theoretical high road to technology and media, offering great resources but mixed real-world application opportunities. Trade schools like DigiPen and the Academy of Art College



focus on specific skills — like game programming or animation modeling, respectively. Other programs, evolving out of multimedia and web design curriculums, tend to be more comprehensive if not yet fully developed (and in the case of community colleges, very accessible if not marginally funded).

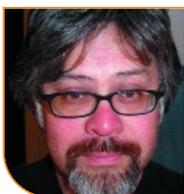
The Academy of Entertainment and Technology at Santa Monica College is a good case study. This community school, in the heart of the entertainment capital that is Los Angeles, receives support from the studio community, has working professionals sitting on its advisory board, and also has a robust intern placement program. Though few schools can teach the type of proprietary applications such as those used in console development, the basic skills of production can be practiced and applied.

Increased enrollment in programs that encourage the integration of the various disciplines, including audio basics for the entertainment industry, is a big plus. A course in ProTools that offers mixing and editing “to picture” is a good starting point, but being able to create and apply audio to branching story lines or user-driven game states should be in the realm

of a comprehensive game design program. The best of these should highlight the differences between traditional and interactive audio. In both commercial and interactive production, learning the tools and the techniques only serves as a foundation for applying the tricks of the trade, tricks that can only be taught by professionals. The senior members of our industry owe it to themselves, and the trade, to get involved if not as part-time instructors, then at least as participants on the advisory boards of these new programs.

Proper facilities with hands-on hardware experience are as important as class lectures. Using authorware such as Director or the more audio-specific Max/MSP can then help the prospective game audio developer to not only score interactive scenarios but also demonstrate them. As for having enough content to use, some schools license the same production libraries — including Sound Ideas for sound effects and APM for music tracks — as the big game production studios. If your company can share something with the educational process, it will contribute to the growth of the business in the long run while creating awareness in new developers.

Being able to prototype a game audio scenario is the only way to gain true development experience. Without this experience, the next-generation developers will have to resort to what they have always done: reinventing the wheel and learning everything from scratch. As we look at the projected growth of game publishing and the development community that must support it, we must reinvest our gains and knowledge into education or be forced to deal with the truckloads of titles that all seem the same. The creative spark that runs dry when we are between technological advances can be kept alive with fresh ideas. Investing in innovation is the best insurance against the inevitable slumps that plague our game industry. If we don’t have the best people, we can’t get the best work done. 



DAVID JAVELOSA | *David currently teaches interactive media at Santa Monica College’s Academy of Entertainment & Technology. A former audio director and game industry specialist, he is working on a revision of his book Sound and Music for Multimedia (Hungry Minds, 1997).*

2 for the Design Process

This month we have two straightforward rules about the design process and the decisions a designer must make early on in game development.

Design Process Rule #1: Design documents should be detailed in inverse proportion to the skill of the team and their familiarity with the genre. This is a common-sense rule: in general the less skilled the game team and the less familiar they are with a given game genre, the more detailed the design document should be.

The rule's domain. This rule applies to the design process itself and the preparation of the design document.

Rules that it trumps. This rule trumps the extremes. Some people insist that no design documents are necessary — ever — while others believe that every single detail must be specified. The former group tends to be novices who have never made a game, vainly hoping to avoid the difficulty of writing a document in the first place. The latter stance tends to come from people who have worked in the software industry, but not on games. They have rightly found that exhaustively specifying the qualities necessary in a piece of software is useful but run into trouble trying to define just how much fun to put in — or how you can tell when it's there in the first place.

Rules that it is trumped by. This rule isn't trumped by other rules as much as certain design processes. With single-person iterative designs, for example, a designer/programmer can start with a simple concept and keep changing, testing, and improving it without using a design document.

The rule is also trumped by situations with external long-distance developers. Even with an experienced team, if the people executing the design are not in daily contact with the designer, more documentation detail is often needed to avoid confusion and wasted time.

Examples and counterexamples. The



Games such as *The Sims* can succeed when designers first understand and then productively question their development constraints.

simplest game design documents I've worked with were a page or two long, for simple single-person development. The largest was an 800-page monster that was a very detailed MMP air combat game intended to be developed by an external company.

Design Process Rule #2: Begin to design by identifying your constraints. The first step in any design should be to identify the critical constraints on that design — what must be done, what should be done, and what cannot be done. Specific areas of constraints can include creative constraints (required game genre or sequel to existing game, the designer's previous experience); technical (the need to use a specific engine or work within the capabilities of a specific programming team); business, sales, or marketing considerations (budget, hard delivery date, license); and personalities (boss's preferences, lead artist's love of anime, producer's fixation on Monty Python). Often, the biggest constraint is budget — usually the vision exceeds the funds.

The rule's domain. This is the earliest

step when working on a design.

Rules that it trumps. This rule trumps not a specific rule but personal ambitions in general. Though you may be determined to create a magnum opus greater than *EVERQUEST*, if you are asked to design a multiplayer game that can be done by a team of eight people in 14 months for \$1 million, you'll have to adjust your aspirations or change your constraints — but don't simply ignore them.

Rules that it is trumped by. This rule is trumped by the exercise of brilliance or persistence. Sometimes you can change the constraints by creating a concept so exciting that you receive additional resources, or by dogged persistence and belief in your own vision.

Examples and counterexamples. Although I advise against blindly battering yourself against unyielding constraints, always question them. When Dreamworks Interactive was just starting up, Steven Spielberg came to us with an idea for a game where you played an ordinary person in a home, dealing with your family, cooking meals, taking out the garbage. We kicked it around, but couldn't get beyond the apparent constraint that few would pay to do that sort of thing, except perhaps the tiny group of people like Steven who were so wealthy that they didn't do those sorts of things on their own anymore. He was very cordial when we told him we thought the audience constrained us. He was even polite enough not to bring it up again when *THE SIMS* came out a few years later.

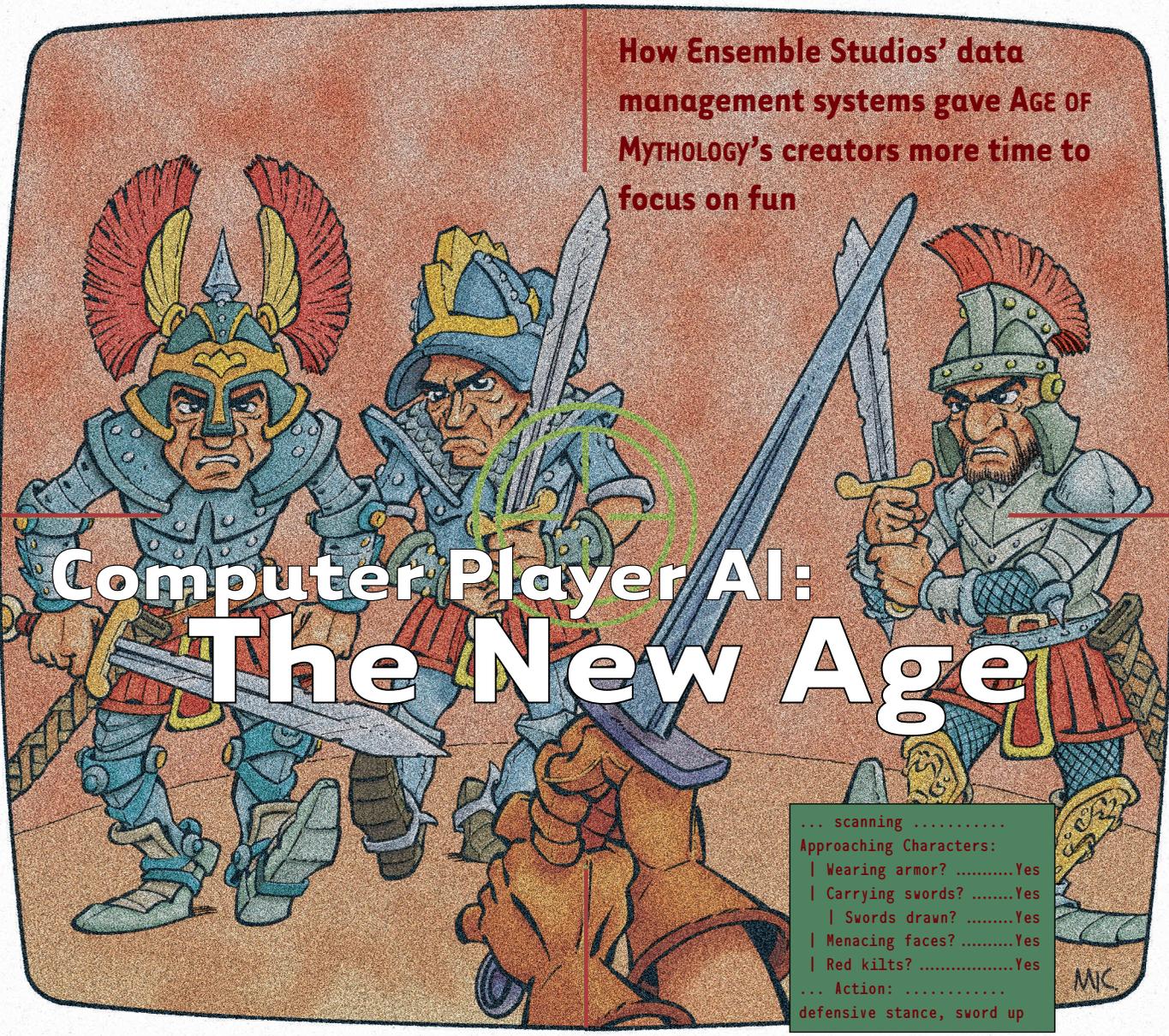
Coming soon. Watch this space for more rules by other designers, plus responses to last month's challenge on consistency. 📧



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

Illustration by Dirk Michiels. Digital Painting by Lieve Huwels

How Ensemble Studios' data management systems gave AGE OF MYTHOLOGY's creators more time to focus on fun



Computer Player AI: The New Age

```

... scanning .....
Approaching Characters:
| Wearing armor? .....Yes
| Carrying swords? .....Yes
| Swords drawn? .....Yes
| Menacing faces? .....Yes
| Red kilts? .....Yes
... Action: .....
defensive stance, sword up
    
```

MIC

How good does AI really have to be? It's a question that few AI programmers stop to ask themselves. "Good" is usually a metric applied to how quickly an AI can completely obliterate your presence in the game. We spend most of our time optimizing the AI's quality without thinking about whether or not the game is more fun as a result.

We've done a lot of RTS games at Ensemble Studios, and RTS games take a lot of AI. We used to be singularly focused on making an AI that could compete with anyone on the planet. While we're still working to make our AIs more and more competitive, a bigger focus for us these days is to make our AIs play a more fun game.

This article takes a look at some of the techniques we used in building the computer player AI for AGE OF MYTHOLOGY (AOM), the third installment in Ensemble's AGE OF EMPIRES

series. None of these implementations provides a single silver bullet for creating nutty-good fun. But, taken as a whole, they illustrate how our methods of data abstraction and representation worked for us. By building a good foundation of knowledge and paradigms for your AI, you will create newfound time and ability to chase that ever elusive fun-factor.

Looking Back to Move Forward

We were fairly happy with the AI on our previous project, AGE OF EMPIRES 2: THE AGE OF KINGS (AGE 2). It was, and still is, one of the few RTS AIs capable of playing fully random maps without any tactical cheating. Nevertheless, the tactical AI code used in AGE 2 was largely the same as that used in the original AGE OF EMPIRES (AGE 1). That AI had several issues we wanted to address in AOM.

First of all, the AI was too atomic in how it represented things. It did an excellent job of remembering exactly what it had experienced. However, due to that overwhelming exactness, it couldn't spend enough time analyzing that data when it came time to do something with it.

Second, AGE 2 used an expert system to drive the strategic part of the AI. Thus, it was fundamentally hard to express an overall strategic plan in the script language when things like dynamically declared variables didn't exist.

And finally, the AGE 2 AI's terrain and movement capabilities were too tightly bound to how the AGE 2 simulation worked. The AI ended up overloaded by the weight of having to know too much about how the simulation really functioned.

First Things First

We were able to anticipate many of the components needed to do the AI for AOM based on our previous experience. We also continually reevaluated those decisions throughout AOM's development. By continuing to improve our abstractions and overall module interactions, we were also able to improve the quality of the final product.

In reality, our AI doesn't use superfancy neural nets or genetic algorithms to make decisions. It uses simple techniques, influenced by the high-level script language content. The underlying data representations are the key; it's those representations that allow us to expose many powerful things in simple ways. We didn't want to get hung up worrying about how to make our AI learn some new trick without first building a proper foundation for how it looks at its world.

Given the realities of today's game schedules, foregoing early architectural work is rarely fixable during a game's life cycle. In our case, the ability to improve the fun-factor of the AGE 2 AI was eventually handicapped by poor data layout decisions we made eight months before AGE 1 shipped. If developers are looking to improve the state, and thus the fun-factor, of AI in games, a larger focus on underlying data representation is needed.

Time to Put up or . . .

So, with the soapbox firmly back under the table, how did we go about implementing the AOM AI to fix these issues we had identified? There were four overriding rules that we used in developing the AOM AI:

Abstract everything. The most important single module in the AOM AI turns out to be the knowledge base, or KB. We shoved absolutely everything our noncheating computer player

DAVE C. POTTINGER | *Dave is the director of technology at Ensemble Studios. He's been there since the days of yore but still manages to end up doing way too much of the AI for their RTS games. Dave enjoys spending time with his family and building insanely sturdy furniture. E-mail him at dpottinger@ensemblestudios.com.*

LISTING 1. SAMPLE OF THE BKUnit PUBLIC INTERFACE

```
class BKUnit
{
    public:
        enum
        {
            cStateNone      =0x00,
            cStateBuilding  =0x01,
            cStateAlive     =0x02,
            cStateDead      =0x04,
            cStateAny       =0xFF
        };
        //Constructors and Destructors
        BKUnit( bool allocateCPInfo );
        ~BKUnit( void );

        //Gets and sets.
        long      getID( void ) const { return(mID); }
        const char* getName( void ) const;
        //PUID.
        long      getProtoUnitID( void ) const;
        //IsType.
        bool      isType( long unitTypeID ) const;
        bool      isAbstractType( long unitTypeID ) const;
        //State (see above enum list).
        BYTE      getState( void ) const { return(mState); }
        //Player ID.
        BYTE      getPlayerID( void ) const { return(mPlayerID); }
        //Position.
        const BVector& getPosition( void ) const;
        //Logical Methods.
        bool      isAlly( const BKB *kb ) const;
        bool      isEnemy( const BKB *kb ) const;
        //Base ID.
        long      getBaseID( void ) const;
        void      setBaseID( long v );
        //Hitpoints and Health.
        float     getHitpoints( const BKB *kb ) const;
        float     getMaximumHitpoints( const BKB *kb ) const;
        float     getHealth( const BKB *kb ) const;
        //Area ID.
        long      getAreaID( const BKB *kb ) const;
        //LOS.
        float     getLOS( const BKB *kb ) const;
};
```

could legally know into the KB. This meant more work on the front end but a lot less work on the back end as we started to build upon that power.

Scripting is good. The AGE 2 expert system gave us a small taste of how successful a script language could be. We decided to build a new interpreted language, called XS, for AOM's AI.

Once done, this work paid large dividends in ease of use and the ability to customize our content.

The AI must have script content to do anything. That may seem trite, but it made a big difference. By forcing this convention, we were able to make the tactical C++ code much more generic, because each strategic script could customize according to its own needs. Scripts could spend their cycles deciding whether to rush or boom to the Fourth Age while the tactical code executed the best building placements. This decision required us to fully embrace the scripting system early on, which meant a better overall interpreted language and toolset.

Eighty percent is good enough. As a programmer, it's hard to ship something that you know can be exploited. But it would have been impossible to catch all of the AI loopholes in a game as large and complex as AOM without several additional man-years of time. In some cases, we skipped making tactical combat improvements in favor of more visible, fun features, such as AI chatting.

The Knowledge Base

The KB is the AI's representation of everything that it knows about the world. It starts with the overall manager class, **BKB**. The KB is a passive device. It serves as the AI's memory, whereas the active components of the AI are handled in another part of the system. The KB can reorganize the data it owns as new objects are discovered, units move around, and so on.

The **BKB** class remembers every unit of consequence that it has ever seen. **BKBUnit** (Listing 1) and **BKBProtoUnit** (Listing 2) represent instance-specific and type-specific data for units in the game, respectively. These objects are kept updated as much as our noncheating system allows.

These **BKBUnit** objects are stored within the **BKBPlayer** structures inside of the KB. Each KB maintains a **BKBPlayer** for every other player in the game, representing his knowledge about that player. The **BKBPlayer** takes advantage of AOM's unit type system to keep handy, precomputed lists of units around. These precomputed lists include:

- Alive units
- Dead units
- Partially-built units
- Type-centric lists of units for all unit types in the game.

The AOM unit type system gives each unit (such as Axeman) a base type in addition to membership in any number of abstract types (such as Military Units). In concert, these subsystems allow the AI to ask questions such as "How many enemy military units have I seen?" or "How many buildings have I lost in this area of the map?" AOM has a very detailed query system built on top of the KB that allows the C++ and script to generically look up any instanced data they want. This abstraction allows the tactical and strategic AI code to do amazing things without any special code in the KB.

KB Example #1: Resource Aggregation

BKB has many specialized data structures to help improve look-up speed and back-end data analysis. A good example is the **BKBResource** class. This class serves to group up various pockets of resources in the world. For example, a herd of deer go into a single **BKBResource**. As a new deer is discovered, we first look to group it into an existing deer herd. If all herds are too far away, on different continents, or otherwise unacceptable, we create a new **BKBResource** instance and link the deer's **BKBUnit** into that.

Because **BKBResources** know which **BKBUnit** entries are inside of them, we can do a very CPU-friendly lazy update of **BKBResources** to maintain their overall resource total, closest dropsite, and centroid. This representation allows us to make more complex decisions when picking a new resource site because our choices are fundamentally limited, as only a small number of **BKBResources** exist. In addition, doing mildly unintuitive things such as computing and storing the closest dropsite during the lazy update also decreases the overall CPU utilization. In turn, this affords us more CPU cycles to spend on decisions that can improve the AI quality and fun-factor.

KB Example #2: Unit Updating

BKB is highly optimized for speed and memory, with most data updating driven by simulation events. Let's look at some of the things that happen when a unit uncovers an enemy Settlement:

- The simulation detects that Player 1's Settlement is now visible to Player 2's Scout.
- The simulation adds the Settlement to Player 2's visible unit system, which pops an event into Player 2's KB.
- A **BKBUnit** entry is created, as this is the first time this unit has been seen by Player 2. If Player 2 had seen this Settlement already, the **BKBUnit** entry would have been retrieved via a simple $O(1)$ lookup.
- This Settlement's **BKBUnit** is fully updated to store current hitpoints, state, and so on.
- The **BKBProtoUnit** for Player 1's Settlement is created if not already in existence. Preexisting or not, it's updated to store common unit data such as inverse maximum hitpoints, line of sight, and armor values. This is possible because those items are the same for all units of the same base type.
- The Settlement's area is updated to reflect the new count of units by relative relationship (self, ally, enemy). This may change the ownership state of this area, as well.
- The KB detects that the Settlement can shoot, so it also updates the danger value for the given area.
- The Settlement is put into a base. Because Settlements are central buildings in AOM, the AI knows it will build a virtual base around this Settlement rather than trying to put it in another base.
- The AI then goes through all other nearby bases to see if

anything in those bases really should belong in this new base. Those that should be moved are moved (and their `BKBUnit` records are immediately updated accordingly).

This is a lot of work to go through during a simple visibility change, but visibility doesn't really change that often. This trade-off is more than worth it given the fast lookups we can do later.

How About Some Action?

The `BAI` class is `BKB`'s more outgoing sibling. `BAI` exists as the big container and overseer of the active parts of the AI. `BAI` is broken down into three layers: Goals, Plans, and Units. In addition to this hierarchy, another useful abstraction is a simple weighting factor that serves to balance between economic and military objectives. Because we put this weighting in early during development, we were able to factor it into every decision the AI made. The script can then do very simple prioritization between the two fairly exclusive parts of gameplay by simply tweaking this number.

AI Plans

Plans are the fancy word for an AI action. `BAIPlans` make up the mid-layer of the AI. They are complex state machines that know how to do tasks such as building structures, attacking a variety of target types, gathering, and the like. Everything the AI does requires a plan. This kept us honest during development, which eventually allowed us to reuse plans in new, interesting ways at the end of `AOM`'s development.

The plans store two types of data: plan-specific data and desired unit types. Plan-specific data may be something like a list of targets for an attack plan (however that list was derived or seeded into the plan) or the type of structure to construct for a build plan. Most all of this plan-specific data is stored in "plan variables" (Listing 3). Both C++ and script code can read plan variables, but some plan variables are only modifiable by the C++ code. This distinction allowed us to put both initialization and run-time variables into the plan variable system, which simplified things for everyone working on the AI. Content developers could also dynamically add their own variables to plans, which allowed them to extend the functionality of plans in their scripts.

This plan variable abstraction also allows the plans to work off of an expected data set without caring how the values were obtained. We did track down lots of bugs during `AOM` where someone would accidentally misuse plan variables, but none of those took more than a minute or two to track down. The big advantage, in terms of AI quality, was that we could write general tactical code in each plan. This tactical code was given many customization points via the plan variable system. This system created better, more fun AI because it was very easy for the content developers to totally customize a plan's behavior.

The desired unit type system was put into place specifically to be a barrier between the script and the actual control of units in the game. From experience, we knew that having the

LISTING 2. THE CLASS DEFINITION FOR `BKBProtoUnit`

```
class BKBProtoUnit
{
public:
    //Constructors and Destructors
    BKBProtoUnit( void );
    ~BKBProtoUnit( void );

    //Update.
    void                update( BUnit *unit );
    //Max HP.
    float               getMaximumHitpoints( void ) const;
    float               getMaximumHitpointsRecip( void );
    //Max V.
    float               getMaximumVelocity( void ) const;
    //Armor.
    const BArmorTable* getArmorTable( void ) const;
    //Damage.
    const BDamageList* getDamageList( void ) const;
    //Damage range.
    float               getDamageRange( void ) {
return(mDamageRange); }
    //Damage type.
    BYTE               getDamageType( void ) {
return(mDamageType); }
    //LOS.
    float               getLOS( void ) const {
return((float)mLOS); }

protected:
    float               mMaximumHitpointsRecip;
    float               mMaximumVelocity;
    BArmorTable         mArmorTable;
    BDamageList         mDamageList;
    float               mDamageRange;
    BYTE               mDamageType;
    BYTE               mLOS;
};
typedef BSimpleArray<BKBProtoUnit*> BKBProtoUnitPointerArray;
```

script try to control individual units was difficult because of interface and latency issues. Plus, a script just isn't very strategic if it's trying to decide what to with Unit #4561.

Scripts and goals were allowed to specify how many units of a given type they wanted to see in each plan. This Need/Want/Max system allowed plans to set the number of units they needed, the number of units they actually wanted, and the maximum number of units they would take. A plan could also simultaneously want more than one type of unit. That particular part of the system turned out to be extremely useful for quickly customizing the AI behavior in our single-player campaign.

Each plan contained its `NWM` values, set via script or high-

er-level tactical code. We also kept a prioritized list of plans at the ready. Every so often, we would run the “Generic Unit Assigner.” Despite the overly pretentious name, the assigner stuffed units into plans based on plan priority and plan NWM values. So, for example, all “need” requests were filled before any “want” requests were considered.

This is a simple algorithm, but we felt it was important to pick some immovable components of the design. This gave the C++ and script code an unchanging implementation upon which it was possible to build. Plans asked for what they wanted and were written to make do with what they were assigned. If the script wasn’t happy with the performance of its attack plans, it had to up the plans’ priority to get more units assigned to them.

The final noteworthy elements of the plan representation are the hierarchy and reuse capabilities of the system. There is one attack plan in the game. It knows how to attack several different types of targets (such as players, bases, individual units, and so on) with a variety of means. This attack plan is reused whenever attacking needs to be done, sometimes as the root plan, sometimes as the child plan. This reuse allowed us to put all of our effort into a single code path, which meant fewer bugs and less overall confusion as the plans interoperated.

Goals Are Part of the Strategic AI

AI goals are super-high-level constructs such as “Attack Player 4” or “Build a forward base in Player 3’s direction.” Goals are entirely created and managed by the script; it’s easiest to think about them as part of the strategic AI. In standard hierarchy fashion, goals know only what the plans tell them and can only tell their child plans to do things. Most of the plan communication with goals consists of sending “success” or “failure” events back to the goals.

The nifty twist comes in at that point. Done properly, the script has already given each goal its preplanned behavior for success or failure. Usually, this behavior is to activate another goal. The script can reevaluate and update those behaviors at any time. But the act of storing them ahead of time both takes advantage of lulls in activity and also increases response time. This goal chaining is one of the best abstractions in the game. If the failure goal of “Build a forward base” is to “Fortify home base,” then we’ve really started to open up a lot of strategic options via a very simple system. Carried through, the simplicity of creating those options means more and better content for your players to experience.

Strategic Attacking

The attack goal knows how to attack a single player ID. It’s the script’s responsibility to set that appropriately. If an AI needs to attack two players at once, two attack goals are created. The attack goal spawns attack plans as needed to make attacks. The attack goal is configurable in ways such as only attacking on a common continent or only attacking every so

LISTING 3. CREATING A SIMPLE MAINTAIN PLAN

```
//XS Script code that uses plan variables to create a simple
//maintain plan to maintain a given number of units in the world.
//=====
// createSimpleMaintainPlan
//=====
int createSimpleMaintainPlan(int puid=-1, int number=1, bool
economy=true,
int baseID=-1)
{
//Create a the plan name.
string planName="Military";
if (economy == true)
planName="Economy";
planName=planName+kbGetProtoUnitName(puid)+"Maintain";
int planID=aiPlanCreate(planName, cPlanTrain);
if (planID < 0)
return(-1);

//Unit type.
aiPlanSetVariableInt(planID, cTrainPlanUnitType, 0, puid);
//Number.
aiPlanSetVariableInt(planID, cTrainPlanNumberToMaintain, 0,
number);

//If we have a base ID, use it.
if (baseID >= 0)
aiPlanSetBaseID(planID, baseID);

//Go.
aiPlanSetActive(planID);
//Done.
return(planID);
}
```

often. Given this high-level method of controlling attacks, we needed a suitably data-driven means to specify with what units to attack. But that method also needed to be responsive to what was going on in the world, without the script needing to micro-manage those changes.

The `BKBUitPicker` was the answer. It’s stored in the KB but exists to serve the AI. The script can create any number of unit pickers. Each can be given a certain set of weighted unit preferences. These unit preferences can be evaluated in combat effectiveness against a set of enemy unit types (dynamically retrieved by a predefined unit query, of course). They can also be weighted by cost and other factors.

Once this abstraction is set up, the AI evaluates it when the goal does its lazy update. The unit picker spits out a prioritized list of units to build in order to meet its objectives. The attack goal uses these results to spawn child plans to construct buildings, train troops, and purchase upgrades for those troops. The net result is that, from a strategic script level, it takes about 25 lines of script code to set up your AI attack personality (Listing

LISTING 4. USING `BKUnitPicker` TO SET AN ATTACK PERSONALITY

```
//Create the unit picker.
int upID=kbUnitPickCreate("Basic Attack UP");
if (upID < 0)
    return(-1);

//Default init.
kbUnitPickResetAll(upID);

//1 Part Preference, 2 Parts CE, 2 Parts Cost.
kbUnitPickSetPreferenceWeight(upID, 1.0);
kbUnitPickSetCombatEfficiencyWeight(upID, 2.0);
kbUnitPickSetCostWeight(upID, 2.0);

//I want to attack with 3 unit types, each made of 2 buildings.
//kbUnitPickSetDesiredNumberUnitTypes(upID, 3, 2, true);

//I want to attack with enough units to equal 20 pop slots, but
//not more than 38.
kbUnitPickSetMinimumPop(upID, 20);
kbUnitPickSetMaximumPop(upID, 38);

//Default to land units that attack land units.
kbUnitPickSetAttackUnitType(upID,
cUnitTypeLogicalTypeLandMilitary);
kbUnitPickSetGoalCombatEfficiencyType(upID,
cUnitTypeLogicalTypeMilitaryUnits);

//Setup our unit preferences. We like lots of infantry.
kbUnitPickSetPreferenceFactor(upID, cUnitTypeAbstractInfantry, 0.8);
kbUnitPickSetPreferenceFactor(upID, cUnitTypeAbstractArcher, 0.4);
kbUnitPickSetPreferenceFactor(upID, cUnitTypeAbstractCavalry, 0.1);
kbUnitPickSetPreferenceFactor(upID, cUnitTypeMythUnit, 0.4);
kbUnitPickSetPreferenceFactor(upID, cUnitTypeAbstractSiegeWeapon, 0.1);
```

4). This allows for really simple attack variation, which makes the game more fun for the human players.

Terrain Analysis

AGE OF MYTHOLOGY has a fairly robust implementation of a terrain analysis system to determine logical map areas. These areas allow us to think about the map in logical terms rather than as a collection of tiles. The basis for this algorithm is nothing more than a simple bounded flood-fill, with a lot of performance and back-end tweaks applied. Most of those back-end tweaks serve to aggregate smaller regions into larger ones.

None of those algorithms are rocket science, but the usage of the data they produce is interesting. Every `BKUnit` knows what area it's in. As it moves, we update unit counts per area. Thus, we can easily see which player or team controls more of the map by looking at the area counts. We can then turn these counts around and use them to figure out what the most vulnerable areas are. Given our noncheating system, this data can

be out of date. In a way, though, that makes the AI more humanlike, which many people find more fun to play against than foes of superhuman skill.

We also tuned the analysis algorithms to produce the right amount of data for an area pathing system. Having too many attack route waypoints can confuse AI logic, whereas having too few looks dumb. In AOM, getting from the AI player's home base to the enemy home base usually yielded five or six waypoints, which felt like the right amount for the game. Tweaking the desired surface area during map analysis will yield more or fewer areas if we need to change that number for our next game.

The area pathing system also let us factor in a lot of AI behavior options. The attack plans can ask the area pathfinder to return routes that enter an enemy base from a different direction than last time. In addition, expected enemy concentrations are easily avoided by factoring area danger into the pathing.

Data That Won't Buy the Farm

The AI in AGE 2 was infamous for its kamikaze farm vendettas. Farms had relatively few hitpoints and couldn't fight back. Thus, they became the perfect low-risk target. Unfortunately, due to AGE 2's unorganized data layout, it was too costly to fix this problem without totally removing the ability to attack farms at all.

Thus, the AOM base construct was born. Bases are logical groups of co-located buildings. The great thing about the base construct is that it immediately limits the AI's search space, be it for placing its own buildings or deciding what targets are viable for attack. In the latter case, the attack plans select a base to attack and then run an area-based attack route to it. Once we know from what direction we'll be entering the base, we can sort the potential targets along that base entrance vector. Having the precomputed list of enemy objects inside the logical base also allows us the time needed to properly handle the farm problem. This feature also makes the AI look a lot more intelligent.

We also use the base entrance vector to do wall-busting. Traditionally, getting an AI to break through walls in a fully random RTS map is hard. But once you have a pregrouped base, we can already determine whether or not the base is walled in. This determination can then be factored into the attack route selection. If we know that an attack route takes us through a wall, we can ask for siege units to be added to our attack plan. That behavior all emerges out of the simple base representation.

The Age of Data Abstraction

This article only scratches the surface of all the code and content behind the AOM AI, but I hope our big push toward fully abstracting all of the game world data has given you a few ideas on how to give your own AI better data with which to work. Once we all have better tools at our disposal, we're one step farther down the road of making an AI that's fun to play against rather than one that just beats up on players. 🤖



Beautiful,
Yet Friendly
Part 2:

Maximizing Efficiency

Last month, in the first part of this series on content optimization techniques (“Beautiful Yet Friendly, Part 1: Stop Hitting the Bottleneck,” June 2003), I reviewed performance at a high level and looked at how level design and environmental interactions affect it.

Since most of the theory behind this month’s article was also explained in the first part, I strongly suggest that readers get familiar with the concepts introduced last month before reading this article.

You’ll need to know when and what to optimize before you can make any use of knowing how to optimize.

Last month, we saw that meshes could be transform-bound or fill-bound. I’ve given a more complete picture of the possibility space here through the generic hardware pipe shown in Figure 1.

If you are data-bound, then the amount of data transferred might also be causing transform problems (too many vertices) and/or fill problems (too much texture data). Data-related problems generally arise through a collection of objects, not by single objects in isolation. If you find that you’re clogging the bus — generally when there’s too much texture data — then you should redistribute your texture and vertex densities across your scene (last month’s article described how to do this). If you are CPU-bound, then it’s out of your hands: the programming team will need to take a hard look at their code.

GUILLAUME PROVOST | Originally hired at the age of 17 as a lowly system programmer writing BIOS kernels for banks, Guillaume has been trying to redeem himself ever since. He now works as a 3D graphics programmer at Pseudo Interactive and sits on his local Toronto IGDA advisory board. You can contact him at depth@keops.com.

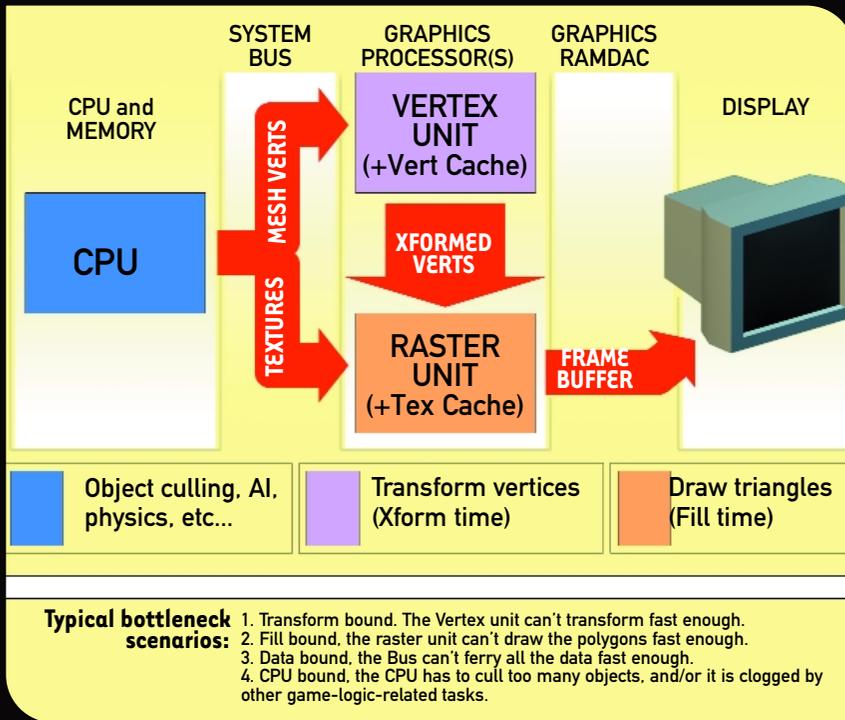


FIGURE 1. A typical hardware rendering pipeline architecture and its associated bottlenecks.

Optimizing Transform-Bound Meshes

If design wants marching armies of zombies attacking the player, you'll need to make sure they don't put the renderer (and artist) on death row by minimizing their transform cost.

We saw last month that the cost of a transform-bound mesh is:

$$\text{Transform Cost} \approx \text{Vertex Count} * \text{Transform Complexity}$$

Hence, we need to reduce the transform complexity or the number of vertices. You can somewhat reduce the transform complexity by plucking out bones you don't really need, but you should consider using a less expensive type of transform first. If you can approximate a morph target accurately enough with a few bones, you'll save on transform complexity. If your engine is optimized for nonweighted vertex blending (where vertices can be affected by only one bone), see if you can substitute your vertex-weighted mesh with

a clever distribution of bones that take no vertex weights. In any case, take the time to consult with the programmers, as they may have insights on better transform techniques you can use to lower your transform complexity.

Welcome to Splitsville

Before you go plucking vertices out of your mesh, I'll let you in on a secret: the vertex counts in your typical modeling package don't reflect reality. As they travel down the pipeline, vertices get split and resplit ad nauseam. Vertex splits adversely affect transform-bound meshes by adding spatially redundant vertices to transform. In theory, vertices can get split as many times as they touch triangles, but in practice, total vertex counts generally double or triple. Keeping this in mind, you can lower this split ratio dramatically and make your mesh a whole lot more performance-friendly without removing a single vertex.

Let's first examine the nature of the splits. As I mentioned last month, graph-

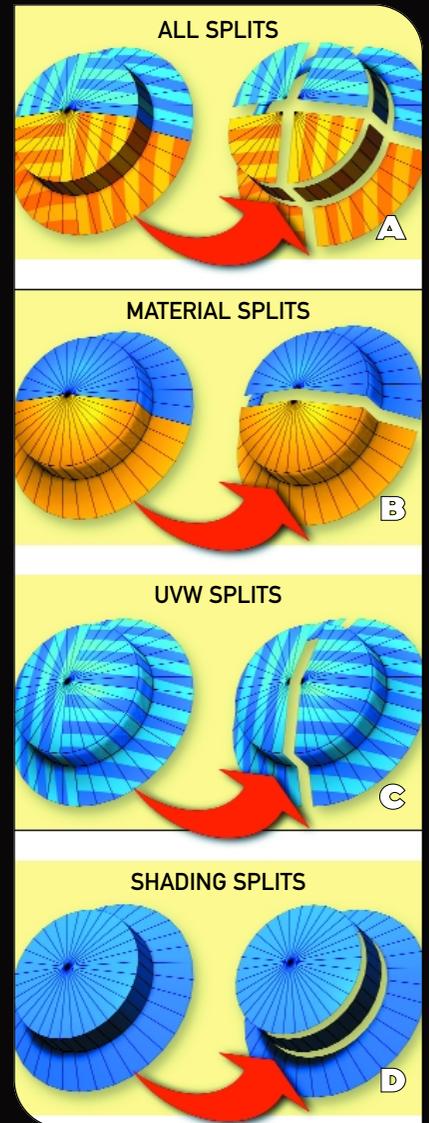


FIGURE 2. Vertex splits accumulate over UV discontinuities, smoothing group boundaries and material boundaries.

ics hardware thinks in terms of surfaces, not objects (that is, the set of all faces in an object that share the same material properties). So the first vertices that get split are those lying on the boundaries of two different surfaces. Think of it in your head as: A vertex cannot be shared across multiple materials (Figure 2b).

Similarly, renderers typically do not allow vertices to share polygons with different smoothing groups, or vertices that

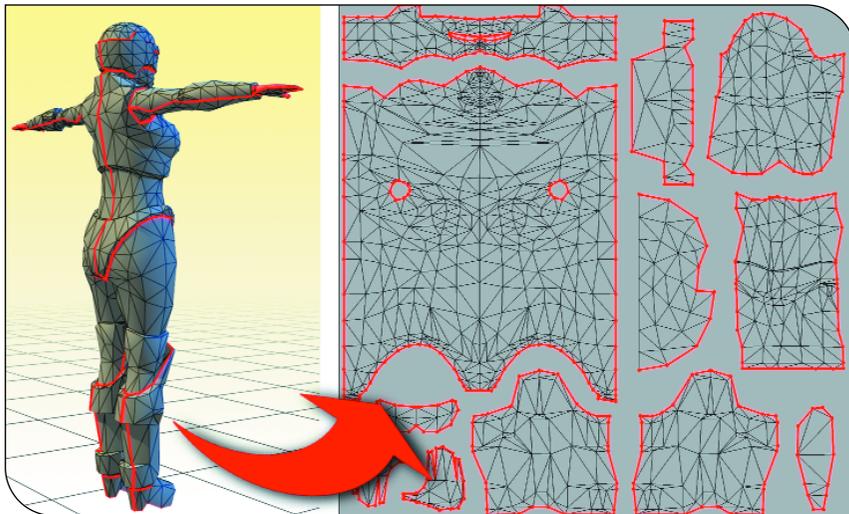


FIGURE 3. The texture is unwrapped on the mesh along a single “sewing” line that wraps the texture like a piece of cloth. This minimizes UV discontinuities (shown here in red) without introducing constraints in the visual look of the mesh.

have different UV coordinates for different triangles. So vertices that lie on the boundaries of two different smoothing groups are split, and vertices that have multiple UV coordinates (which lie on the boundaries of discontinuities in UV space) will also cause splits (Figures 2c and 2d). Moreover, if you have objects with multiple UV channels, the splits will occur successively through every channel.

There are several simple ways to minimize individual types of splits. Intelligently combining and stitching textures together, for example, can help minimize material-based splits.

UV space discontinuities tend to be a bit trickier. Mapping an element without any UV break means that you’ll have to find either an axis of symmetry or at the very least a “wrapping point” on your mesh.

If you can get away with using mapping generators, such as planar, cylindrical, or cubic mappings, you can minimize or altogether eliminate UV space discontinuities. Ball-jointed hips and shoulders, for example, can make the resulting arm and leg elements ideal candidates for such techniques.

If you need to split the mesh in UV space, both 3DS Max 5 and Maya have

elaborate UV-mapping tools that permit you to stitch UV seams in order to minimize the damage. (Maya even has a UV-space vertex counter, which should reflect the number of vertices in your mesh after UV splits.) It’s generally well worth spending the time to optimize your mapping in UV space, since it will also both simplify your texturing pass and minimize the texture space you will actually need for the object. When no axis of symmetry existed, we found that treating the texture as pieces of cloth that you “sew” up worked well to minimize UV splits when texturing humanoids (Figure 3).

If you are building a performance-critical mesh, it’s probably best that you

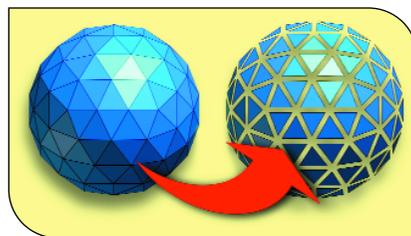


FIGURE 4. Flat shading causes every face in a mesh to belong to a separate smoothing group, causing a worst-case split scenario to occur. Avoid at all costs unless specifically supported.

fine-tune and optimize the smoothing groups by hand. Remember that the goal isn’t to minimize the number of different smoothing groups, but rather the number of boundaries that separate those smoothing groups. You can also fake smoothing groups by using discrete color changes in the texture applied to it, avoiding splits altogether, although this may not result in the visual quality you are attempting to achieve.

Another way to look at it in the big picture is to “reuse” vertex splits. For example, I said earlier that renderers allow one material per vertex and one smoothing group per vertex. In other words, if you have a smoothing group and a material ID group that occupy the same set of faces, they’ll get split only once. The same goes for UV discontinuities: if they occur at smoothing group boundaries, then they won’t cause an extra split to occur.

For the record, if your mesh is definitely transform-bound, then it is generally more important for you to save on vertex splits than to save on texture memory. If that means authoring an extra texture for the mesh in order to get rid of individual diffuse color-based materials or UV breaks, then it’s a fair trade-off.

This brings us to normal maps and the general (and increasingly popular) concept of using high-detail meshes to render out game content. Normal maps are textures for which every texel represents a normal instead of a color. Since they give extremely fine control over the shading of a mesh, you can replicate smoothing groups and add a whole lot of extra shading detail by using them. Since normal maps are generally mapped using the same UV coordinate set as the existing diffuse texture, they do not cause extra vertex splits to occur, and are in effect cheaper for transform-bound meshes — and much better looking — than smoothing groups.

Unfortunately, normal maps cannot really be drawn by hand; they require specialized tools to generate them, and also require higher-resolution detail meshes if you want to take full advantage of their potential. Because of the pixel oper-

ations involved that are required to support them, they are also not supported on all hardware platforms.

Overall, absolutely try to avoid checkerboard-like material switches, where you consistently cycle between materials. Unless your programmers specifically support it, also avoid setting whole objects as flat-shaded by having individual faces be a different smoothing group (Figure 4).

Helping the Stripping Process

When I originally set out writing this article, I naively thought I could safely cover solid guidelines that covered all mainstream console systems and all recent PC-based graphics cards without encountering critical system-specific guidelines. I was overly optimistic.

Some systems don't support indexed primitives, and some don't have a T&L transform cache. In either case, your surfaces' transform cost will be significantly affected by their "strip-friendliness." If your hardware does support both, then strip-friendliness is less of a performance issue.

A triangle strip is a triangular representation some systems use in order to avoid transforming a vertex multiple times if it's shared among one or more triangles. In a triangle strip, the first three vertices form a triangle, but every successive vertex also forms a triangle with its two predecessors. When graphics processors draw these strips, they only need to transform an additional vertex per triangle, effectively sharing the transform cost of the vertices with the last (and next) triangle.

Stripping algorithms close a strip (effectively increasing transform time) when there are no vertices they can choose in order to form a new triangle. This typically happens at tension points (Figure 5), where a single vertex is shared amongst a very high number (eight or more) of triangles. (Certain renderers support what are called triangle fans. Fans make tension points very efficient, but given that cur-

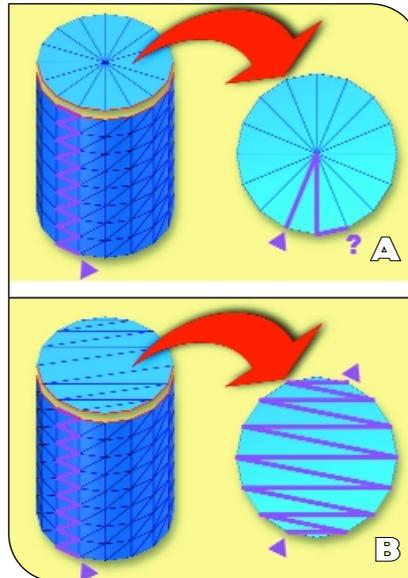


FIGURE 5. At top, a cylinder cap is improperly triangulated, causing the strip to "break" very early. The strip cannot cross to the main body of the cylinder because of smoothing group splits. At bottom, the cap is retriangulated properly, and fits completely in a single strip.

rent hardware only supports one type of primitive per surface, they tend to rarely be supported in practice.)

Since tension points are always connected to a series of very thin triangles, avoiding sliver triangles and distributing your vertex density as equally as possible on the surface of your mesh will generally help the stripping process.

Most good triangle-stripping algorithms will automatically retriangulate triangles lying on the same plane, but they cannot reorient edges binding faces on different planes. You should verify these details with the programmers.

Transform-Bound Meshes Conquered

Knowing about all these technical details can make a transform-bound mesh up to three times more efficient if you're smart about what you're doing, but it's still a lot of work. Always ask yourself whether you need to optimize a

mesh before you dive into the hard work. Otherwise, use these techniques opportunistically. In the end, having a tool that helps visualize where vertex splits occur is tantamount to building truly optimized meshes. As a summary of things to look out for, here's an optimization checklist for transform-bound meshes:

- Build one or more LOD (level of detail) meshes for the object.
- Use as few bones and vertices as you can, and try to decrease the transform complexity.
- Use as few material surfaces as you can get away with; consider texturing your mesh instead of using several different diffuse colors.
- Use UV generators to minimize UV discontinuities.
- Get rid of smoothing group breaks you don't really need, or use discrete color changes to fake them, or use a normal map.
- Match the remaining material boundaries, UV-space boundaries, and smoothing group boundaries.
- Validate your invisible edges and look out for tension points.
- Avoid sliver triangles and try to make the vertex density as uniform as possible across the surface of the mesh.

If you think that your mesh is fill-bound instead of transform-bound, then do not do any of the above. Combining materials into a single texture applied to a fill-bound mesh, for example, might actually hurt your performance by causing cache misses to occur more frequently, so fill-bound meshes warrant separate optimization considerations.

Optimizing Fill-Bound Meshes

We saw earlier that the cost associated with drawing fill-bound meshes was a function of three things:

$$\text{Fill Cost} \approx \text{Pixel Coverage} * \text{Draw Complexity} * \text{Texel Density}$$

You can't make your walls any smaller than they are, but you should avoid over-

laying several large surfaces within the same visibility space. A typical example of this would be to have an entire room's wall covered with an aquarium (the back wall and the glass window create two layers), or successive sky-wide layers of geometry to simulate a cloudy day.

Transparent and additive geometry tend to accumulate on-screen, potentially creating several large layers of geometry the renderer needs to draw, thereby creating a fill-related bottleneck.

If your export pipeline supports double-sided materials, be wary of using them arbitrarily on large surfaces; you can easily double your fill-rendering costs if you are forcing the render to draw wall segments that should be culled. On some platforms, back-face culling is not an integral part of the drawing process, and culling individual polygons becomes a very expensive task; if you are authoring content for such platforms, you should ensure that walls that don't need back faces don't have them.

The bigger the triangles, the less texture space you want to address. Unfortunately, in practice, meshes that take up the largest portion of screen space also tend to also gobble up the most texture space, and so they are prime targets for being fill-related bottlenecks. There are two things you should do to minimize your texture space: make sure you are using and generating mip-maps, and choose your texture formats and size intelligently.

Table 1 illustrates savings you can achieve by making smart choices about your texture formats. Note that if your textures are smaller than 32×32 texels, it's probably not a good idea to palletize them, since the cost associated with uploading and setting up the palette is larger than just using the unpalletized version. If your hardware supports native compression formats, such as DXT1 (DirectX Texture Compression), it's a good idea to use them over palettes.

If you can get away with using diffuse colors only on a fill-bound surface, so much the better. On several platforms, drawing untextured surfaces is faster than drawing textured ones.

I mentioned earlier that it was general-

Size/Format	16-color PAL	256-color PAL	16-bit RGB	32-color ARGB	DXT1 RGB
32 X 32	Do not use	Do not use	2K	4K	1K
64 X 64	2K	4K	8K	16K	4K
128 X 128	8K	16K	32K	64K	16K
256 X 256	32K	64K	128K	256K	64K

TABLE 1. This table illustrates simple savings you can do by making smart choices about your texture formats.

ly a fair trade-off to sacrifice texture space in order to prevent UV splits in transform-bound meshes. When your mesh is fill-bound, however, the contrary rule applies: if splitting the vertices in UV space will help you save texture space, it's also a fair trade-off.

Finally, conservative decisions on the nature of the materials you apply to fill-bound meshes pay off in performance. The number of texture passes and the complexity of their material properties is always the biggest factor at play when dealing with fill-bound surfaces.

Texels Miss the Boat

Some of us deal with the *crème de la crème* when it comes to hardware, but the vast majority of us need to contend with market realities. In the console market, teams get to push a system to its limits, but they are also stuck with those limits for a long time.

If you count yourself in that situation, then chances are you need to take something called texel cache coherency into account. Here's how it works.

Graphics processors typically draw triangles by filling the linear, horizontal pixel strips that shape them up in screen-space. Almost all current hardware can do this by "stamping" several pixels at a time, greatly decreasing the time it takes to fill the triangle.

For every textured pixel the card draws, it needs to retrieve a certain amount of texels from its associated texture (since the pixels are unlikely to fall directly on a texel, renderers typically set up video hardware for bilinear filtering, which fetches and blends four texels for each texture involved). It does this through a texel cache, which is basically a

scratchpad on which the card can paste texture blocks. Every time the card draws a new set of pixels it looks into its cache. If the texels it needs are already present in the scratchpad, then everything proceeds without a hitch. If some texels it needs are not in the cache, then the card needs to read in new texture chunks and place them in the cache before it can proceed with drawing. This is called a texture cache miss.

A good texel cache coherency means few texture cache misses occur when drawing a surface. A bad texel cache coherency will significantly increase the time it takes to draw a surface. Most PC-based systems and a few of the current high-end consoles will automatically ensure a good texel cache coherency by choosing the proper mip level at every pixel they draw. But other systems rely on the fact that the texel density across the surface area of a mesh in geometric space is constant for their mip level choice to be correct.

On such systems, nonuniform texel densities will cause the card to "jump" in texture space from pixel to pixel. This can cause severe texture aliasing problems and will also consistently cause texture cache misses to occur as the card tries to fetch texels that are not in its scratchpad.

As an artist, you can solve both those visual artifacts and performance problems by ensuring you uniformly distribute texel density across your mesh (Figure 6). You can do this by ensuring that the size and shape of your faces in UV space is roughly proportional to their counterparts in geometrical space. This is a concept that makes sense from an artistic perspective as well: if a face is bigger, it should get more texture detail (a larger UV space coverage) than a smaller one.

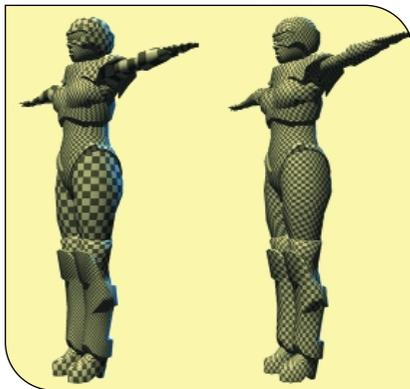


FIGURE 6. At left, nonuniform texel densities will create visual artifacts on certain platforms. At right, the texel density is uniform as a function of geometric space. If this was a fill-bound mesh, the unequal texel density would also cause cache misses to occur on certain platforms, effectively increasing the total fill cost of the mesh.

The concept extends to objects too: if an object is smaller, it's likely to be smaller on-screen as well, and should get a smaller (less detailed) texture.

Fill-Bound Surfaces Conquered

Following is a list of things to do and look out for when constructing fill-bound geometry:

- Build mip-maps for all textures.
- Shy away from large surfaces with complex material properties, such as bump maps and glossy materials.
- Don't overlay several very large transparent or additive layers.
- Don't make large wall/ceiling segments double-sided unless you absolutely must. If your engine doesn't support back-face culling, make sure to get rid of large, unnecessary back faces.
- Choose your texture formats intelligently to save texture space. If you do not have access to compression formats such as DXT1, see if you can't palletize textures.
- Use small texture swatches or diffuse materials instead of larger textures, even at the expense of vertex splits.
- Tweak your UV maps to distribute

your texel density as uniformly as possible across the surface.

The good news about fill-bound surfaces is that, although adding more vertices probably won't help, it probably won't make much of an impact until your vertex density is high enough for your mesh to become transform-bound. (However, very large polygons can on some systems trash the texture cache, effectively increasing fill time. In such cases, tessellating the polygons will actually help.)

Be Fruitful and Optimize

If your head is spinning by now, remember Douglas Adams's motto: Don't panic. Although there is a lot more to performance-friendly content than meets the eye, building efficient content can become an intuitive, natural process with practice.

Whether they are vertices, texels, objects, or textures, it's more about uniformly distributing them than about plucking out detail. This is a very powerfully intuitive concept: things that are smaller on-screen should get less detail than things that are bigger on screen.

Programmers can always optimize their code to go just a little bit faster. But there's a hardware limit they can never cross without sacrificing visual quality. If you are pushing the limits of your system, chances are that it is your content — not code — that drives the frame rate in your game. 🦄

ABOUT THE ILLUSTRATION

The mesh of the character on page 30, consisting of 2283 vertices (after mesh conversion for in-game readiness), was constructed with real-time constraints in mind and makes use of a lot of pointers discussed in this article: texture seam reductions, uniform texel density, minimal material changes, smoothing group optimizations, and others.

Image created by Danny Oros.



Big Huge Games' RISE OF NATIONS

The name of our company is a summation of our corporate attitude: aim for the top, but don't take yourself too seriously along the way. In this case, "aiming for the top" meant putting together a company and a game designed to go head-to-head in one of the most competitive and resource-intensive segments of the PC marketplace: real-time strategy games. To succeed, our first game needed all the fun, depth, and polish of products that enjoyed bigger budgets and more manpower due to their recognizable franchises. Since they don't give out Game Developers Choice Awards for "Best Game Made With Fewer Than 30 People," we had to find ways to work both harder and smarter if we wanted to achieve our goals.

Big Huge Games was formed in early 2000 by a core team who had worked together for close to 10 years, creating best-sellers such as COLONIZATION, CIVILIZATION II, and ALPHA CENTAURI. This history of successful strategy games allowed us to go to publishers with a convincing pitch for a next-generation RTS. Although we understood the issues involved in creating turn-based games, almost all of the areas where we failed to address risks adequately involved areas where we had minimal experience, such as multiplayer match-making and making linear sin-

gle-player campaigns. In addition to these, we also stumbled in some areas that were unique to our situation and company culture.

What Went Right

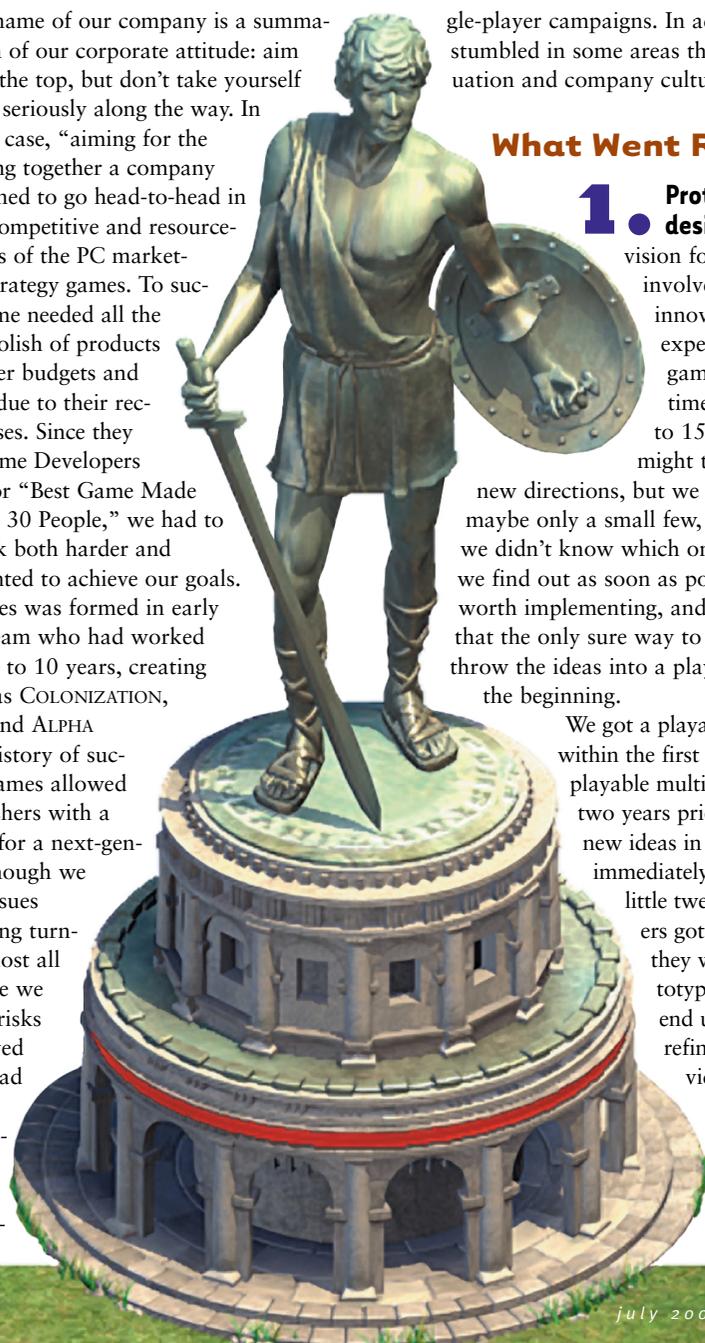
1. Prototype method of game design.

Part of the core vision for RISE OF NATIONS involved introducing gameplay innovations inspired by our experience making turn-based games into the "classic" real time strategy mix. We had 10 to 15 "wild" ideas about what might take realtime strategy in

new directions, but we knew that only some, maybe only a small few, were going to work, and we didn't know which ones. It was essential that we find out as soon as possible which ideas were worth implementing, and we knew from experience that the only sure way to accomplish this is to throw the ideas into a playable prototype right from the beginning.

We got a playable solo prototype running within the first month, and a fully playable multiplayer version more than two years prior to ship. We could throw new ideas in and see the results almost immediately: some concepts needed a little tweaking to be fun, while others got trashed almost as soon as they went in. The value of prototyping is that core concepts end up being continuously refined over years, while providing lots of time to balance the game.

As part of the prototype approach to design, we make sure that everyone in the



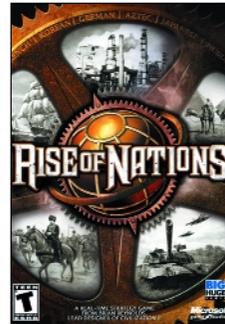


company is playing the game on a regular basis. After each daily play session, a member of the design team compiles everyone's feedback and sends a summary to the rest of the designers. However, we found that we had to be willing to wade through some resistance to new features or gameplay tweaks. People would get very attached to certain strategies for playing or even just conventions of RTS games of which they were hesitant to let go.

2. Choosing the right publisher. Over the years, the production values and polish levels on RTS games have risen along with the popularity of the genre. Given the scope of the competition we were up against, a major concern for us was finding a publisher who would be willing to invest the resources necessary in a new company to produce a product that could go head-to-head with the “big boys” — in essence, finding a publisher who shared our culture and values. Our task was made somewhat easier by the fact that the core team had already notched a couple of million-sellers with other companies, but we still had difficulty in selling publishers on our business model.

In early 2000, when the company began, the industry was in the throes of online mania and at the height of the Internet bubble. The “smart” money was flowing to online game sites and massively multiplayer titles. However, in our pitch meeting with Microsoft, we were impressed with their approach: when we asked them which of our five proposals they were most interested in, they just asked us which game we'd be most interested in making. They seemed more interested in the team than in the specific proposal, which in our experience is a great approach to produce top-quality games.

Once we signed on board with Microsoft, we were amazed at the level of support they gave us. When we thought of Microsoft, we took their marketing and sales capabilities for granted, but we were equally impressed by the quality of their development support. Two key groups really helped us polish our game: the

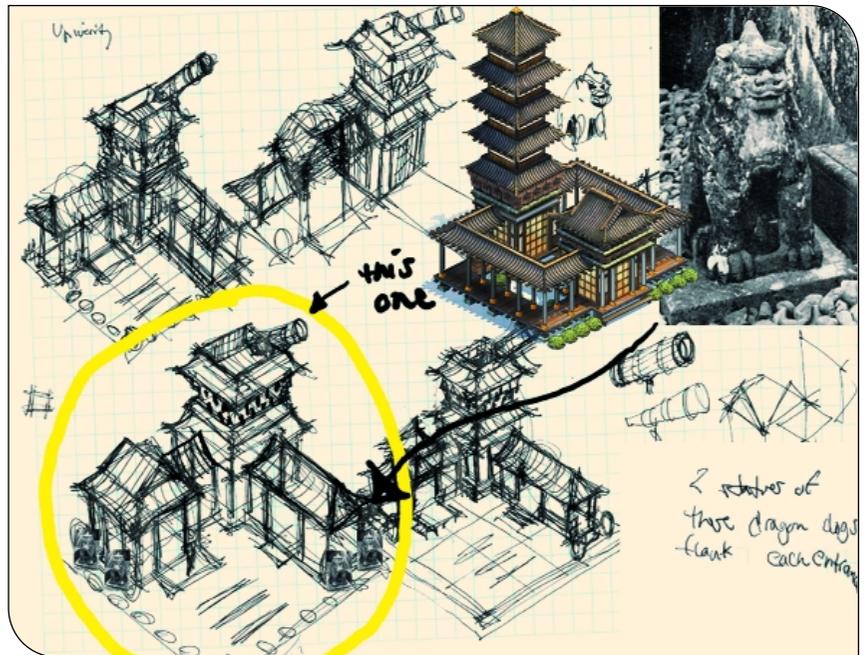
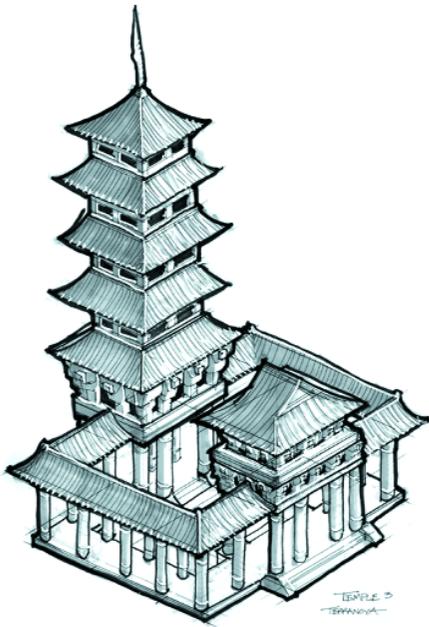


GAME DATA

- PUBLISHER:** Microsoft
- NUMBER OF FULL-TIME DEVELOPERS:** 26
- NUMBER OF CONTRACTORS:** 16
- LENGTH OF DEVELOPMENT:** 3 years
- RELEASE DATE:** May 20, 2003
- TARGET PLATFORM:** PC
- DEVELOPMENT HARDWARE:** WinXP PC
- DEVELOPMENT SOFTWARE USED:**
 - Boundschecker, Altova XMLSpy, Araxis Merge, MacroExpress, PCLint, 3DS Max, Character Studio, MS Developer Studio 6.0, Perforce Source Control, Koreax Incredibuild, Visual Assist, Workspace Whiz, Alexsys Team, Adobe Photoshop, Adobe Premiere, Intel VTUNE
- PROJECT SIZE:** *.C, *.CPP, *.H:
 - 1,721 total source files,
 - 837,939 total lines, 24,610,223 total bytes;
 - *.BHS: 46 total files, 24,330 total lines, 966,289 total bytes

TIM TRAIN | As vice president of operations and development, Tim heads up the internal development of Big Huge Games. With 12 years of leadership and team experience on numerous landmark PC titles beginning with CIVILIZATION I, Tim has worked in every genre of computer games. Tim served as executive producer and designer on RISE OF NATIONS.

BRIAN REYNOLDS | Brian is president of Big Huge Games and a 12-year industry veteran. Honored by PC Gamer as one of 25 “Game Gods,” Brian has masterminded the design of numerous hit strategy games, including CIVILIZATION II and ALPHA CENTAURI, and now RISE OF NATIONS.



caption here

play-balancers and the usability labs. Throughout much of the last eight months of the project we had four to six full-time play-testers assigned to the project whose sole job was helping to balance the game. These guys were expert-level RTS players who could smoke the designers after very little time with the game. They helped us find and fix all kinds of broken strategies and degenerate gameplay, and ensured a much more balanced game for hardcore players right out of the gate.

The usability labs took care of the other end of the spectrum, the casual players who aren't as familiar with the ins and outs of RTS games. Between the various tutorials, core gameplay, and the Conquer the World single-player campaign, members of Big Huge took up five weeks of the usability labs as we watched beginner-level players struggle through basic game concepts. Our programmers were there in the labs, coding changes on the fly, able to put a new version up for the next subject. Usability's input resulted in hundreds of changes to the game, making it more streamlined and easy to jump into. It's hard to overstate the contribution both the play-test and usability groups made to the final product.

3. Disciplined hiring process. We started the company with a proven method of developing strategy games through prototyping, and soon afterward we had a publisher that shared our vision and complemented our strengths. What we didn't have, and what would certainly be the biggest single factor in achieving our goals, was a full team.

From the start of the company, we took great care in selecting our staff, adopting an interview system that we thought worked well for Ensemble Studios. All candidates that make it through

two rounds of phone interviews are brought to Big Huge and interviewed by every person on staff. After each interview the company meets and discusses the candidate's strengths and weaknesses, at which point anyone can veto a hire for any reason.

Although this process has gotten more time-consuming and difficult as the company has grown, the end result has been well worth the effort. A major advantage of this system is ensuring that every new hire can work and play well with others; if they can impress through multiple interviews with a diverse set of people, then we can have confidence that they will fit in from day one. Perhaps most importantly, it ensures that none of our full-timers has the experience of being introduced to someone they've never met before in their life as "... And here's who you'll be working with on such-and-such for the next year."

4. Developing powerful in-house tools. Our programmers worked from the philosophy that taking a little extra time initially to develop a good tool or algorithm pays off manyfold in time saved over the course of the project, while improving the quality of the final product. We also learned not to rely exclusively on one particular tool but rather to use an array of tools to help narrow down a problem. Following are some of the internal tools and techniques which paid the biggest dividends for us:

The section profiler. Our lead programmer, Jason Coleman, created an interactive visual profiler, which helped us identify the correct time segments to bring optimization resources to bear, particularly when trying to identify intermittent spikes in performance that would ordinarily be averaged



out when running a profiler over many game frames. Our programmers mark the beginnings and ends of key sections of our code as belonging to one of about 20 color-coded categories (such as render units, network, pathfinding, and so on). Then, as the game runs, the programmers have access to an interactive graphic window with slider bars for time and scale. The profile chart changes color each time the program moves from one section to another, and longer times in a particular section of course result in longer blocks of a particular color. It is easy to spot spikes in a particular section and then, using our “recorded game” feature to repeat a game precisely, a more powerful profiling tool such as VTune can be brought to bear on just that particular time segment, thereby avoiding the “averaging out” effect. The section profiler also helps us spot sections that are being entered too often, even if not for very long.

The parameter window. Graphics programmer Jason Bestimt created an interactive “parameter window” module, which allows our programmers to register as many variables as they’d like as parameters, which can then be interactively controlled during the game with their choice of slider bars, combo boxes, or edit boxes, using a special pop-up console without causing performance degradation. Being able, for instance, to pull up an interactive page that controls all of the render states for any desired graphics element made for great progress on special effects. For example, the nuclear blast effect could be fine-tuned without having to repeatedly recompile (or even rerun) the game: the programmer and artist just sat there and pulled on the slider bars until it looked just right.

The Const System for multiplayer. One of the great nightmares of creating multiplayer strategy games is keeping the game world synchronized across each of the player’s machines. Game code and random number generators must run in virtual lockstep across every machine in the game, or the whole game world shatters and goes out of sync — the multiplayer programmer’s worst-case scenario which effectively ends gameplay. Interaction with the outside world (such as the commands players give to their units) must be carefully propagated to all machines before they can be safely executed or even safely “seen” by code that can write to the game state. The smallest unintentional bypassing of this rule can result in disaster (such as a graphics routine that accidentally uses the game-side random number generator, or an input routine that directly affects the game world without passing it through the network protocols).

To avoid most of the potential catastrophes of this type we created the “Const System,” essentially a compiler-assisted firewall between the game-world side of the code and the I/O (graphics and interface) side of the code. There’s game-side data (the simulation) and non-game-side data (everything else). Game-side is allowed read-write access to itself but write-only access to the non-game-side (rendering, for example). Non-game-side is allowed read-write access to itself but read-only (const) access to the game-side (user input isn’t allowed to

directly modify the game-state). The real compiler tricks involved the fact that C++ doesn’t have an inherent notion of write-only (new standard, anyone?). Also, once this was worked out, all remaining issues involved either the occasional, foolhardy overriding of the system, or bugs such as uninitialized data or memory overwrites. Two sets of macros (one each for game and interface access) made this scheme mostly transparent to programmers. The end result was that many of the potentially thorniest multiplayer bugs became easy-to-find “compile errors” instead of nearly impossible-to-find intermittent out-of-syncs.

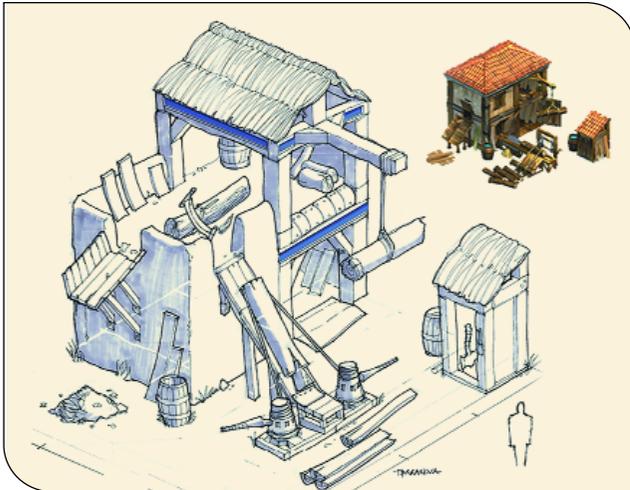
5. Great third-party productivity tools. One of the great life-changers for our team was Xoreax’s Incredibuild tool. Essentially it lets us turn every machine on our company network into a vast compile farm — the tool automatically makes use of free cycles on everyone’s machines to compile our game at jaw-dropping speeds. Even when RISE OF NATIONS was in gold release, our entire game could compile, including all of the libraries, in just under two minutes. Optimizations added an entire extra 15 seconds to the process. Linking the code, the one step which must be performed entirely on the programmer’s own machine, took an extra 45 seconds. So, in other words, the final release of RISE OF NATIONS that we delivered to Microsoft probably compiled and linked in around three minutes.

The ability to compile (and recompile) the code so quickly led not only to greatly speeded debugging and development, it also resulted in much cleaner code: programmers no longer feared changing header files with the half-hour-plus recompile formerly associated with such an action, so they no longer felt tempted to resort to messy workarounds and hacks just to avoid a full recompile. An amusing side-note is that programmers who had structured their lives around using long compiles to grab drinks, chat, and play chess suddenly had to rethink their whole day.

Other tools well-loved by our programming team include the



caption here



caption here

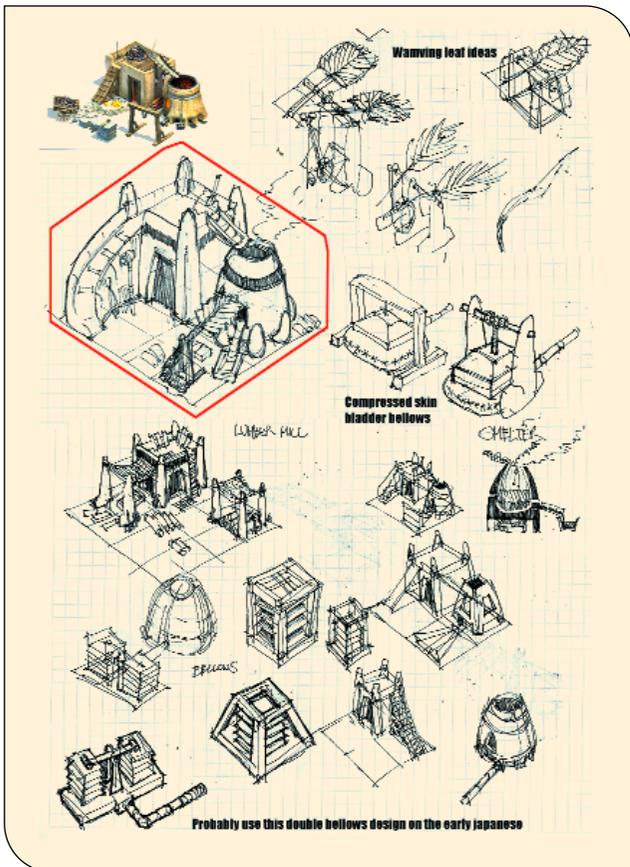
Visual Assist and Workspace Whiz add-ons to Microsoft's Developer Studio. These two tools add a ton of little improvements to the development environment that end up completely changing the way programmers use Dev Studio. Features include toggling directly from .CPP to .H files, opening any file in your workspace (without needing to provide a path), and automatic grammar correction. Our programmers always comment about how sad they are when they have to debug on a machine without these tools installed.

From the standpoint of task organization, the lifesaver for us was Alexsys' Team software. This system replaced our "Post-It note" method of project management with an extremely configurable setup and easy access to tasks. One of the greatest features of Team is its capacity to send e-mail alerts whenever a task is created or modified. As a company we use e-mail a lot, and having a project management system that essentially forced everyone to stay current with tasks was invaluable.

What Went Wrong

1. Not listening to all the other Postmortems ever printed in *Game Developer*. The Postmortems are the most widely read feature in *Game Developer* around Big Huge, and yet somehow we still managed to make many of the mistakes developers are cautioned against in these pages. We underestimated the amount of coding time necessary, which resulted in an extremely overworked programming staff. We misjudged the amount of revision time that we'd want for various systems. We overloaded our lead programmer such that he became the bottleneck on a number of critical systems, including multiplayer and matchmaking. Most of this could be traced back to simply not hiring enough programmers early in the project, and was compounded by lack of scheduling and technical oversight. We also never knew what was "enough" — since this was our first project in the RTS world, we were desperate to cram everything in that we could think of. For the next project, we will certainly hire more programmers and not schedule our lead programmer for anything other than management and support, with the expectation that he will have some flexibility to jump in and help out wherever it's needed. We also expect that we'll have a much more straightforward perspective on scheduling to meet our goals.

Another classic blunder (comparable to getting involved in a land war in Asia) was in undervaluing single-player tool creation. We always assigned our most recent programmer hire at any time to be the "scenario tools" guy, meaning we not only always had our least-experienced team member doing that work, we also had no continuity since we'd pass the torch each time we hired someone new. The editor also suffered grievously from several revamps of the game's terrain system and other parts of the engine. Hence, we had a great deal of difficulty creating single-player scenarios, and we were very fortunate to have the Conquer the World campaign turn out as well as it did. In the last few weeks of development, programmers Ike



caption here

Ellis and Scott Lewis finally whipped the editor into shape with some intense hours and smart coding, but for the next game we will have someone working on this module early and throughout the project. This task will also be easier because we'll be working from a more mature engine.



2. No clear idea of the kind of game we were making. In the first year, both we and the publisher waffled back and forth on whether we should be doing a “classic” RTS game, a completely new kind of strategy game with some real-time elements, or something in-between. The prototype would swing back and forth between those two poles. Eventually, external events (such as the release of *Empire Earth* and Microsoft’s purchase of Ensemble Studios) made it clear that a straight-down-the-line “classic” RTS was not the right game to be working on, but not before several months of work on art and game design were wasted going back and forth.

For the next game, we have a much better sense of what our style of game should encompass — epic scope, strategic depth, and large armies clashing — and hopefully we won’t have to be so concerned about standing in the shadows of the other giants of the genre. We’ve also got a much stronger sense about what works and what doesn’t in an RTS game.

3. Hard time finding the look from the art perspective. Partly because of the preceding problem, we took longer than usual to nail down an art look for the game. Among other issues, it took us some time to decide whether we’d be fully 3D. The rest of the market was going full 3D, but we really loved the detail and crispness that 2D offered for things such as building graphics. The only engine feature we would give up to go 2D was the ability to rotate the camera, which we’d never found to be very useful in RTS games anyway. However, there was also a lot of hand-wringing about whether we’d be considered behind the curve graphically.

We did numerous tests with 3D and discussed the issue with marketing, but in the end we went with a 3D engine that utilized 2D for buildings. We’ve been happy with how the game looks and think the high detail on the buildings adds a lot to the world. Next time, we’ll do a lot more prototype art and concept work before going into full production, and we’ll be more confident diving into a fully 3D world off the bat. Having an experienced, full staff will also help with this issue — at the beginning of *RISE OF NATIONS* we made do with just a couple of artists.

We were also faced with the classic dilemma of doing a history game — it’s hard to differentiate yourself from earlier products when you are drawing from common source material. A pikeman looks like a pikeman, no matter how radical an approach you try to take. For the most part, we attacked this issue on the marketing end. We focused on creating screenshots from the later eras of the game and highlighted the diversity in cultural art sets from

nations that hadn’t been covered as much in other products. Art leads Bill Podurgiel and Ted Terranova worked hard to create units and buildings that were both realistic and varied, ultimately helping to help differentiate the look from that of other products.

4. Solo scenario meltdown. When we started work on *RISE OF NATIONS*, we assumed that the single-player game would feature classic RTS-style linked scenarios that followed a nation’s history over time. However, we started work on those scenarios and found that they just weren’t particularly appropriate for our subject matter — it’s not a lot of fun to take a game about all of history and then constrain players to a smaller canvas and scope. This approach also did not play to our strengths as developers; we are more interested in creating open-ended and infinitely replayable experiences than in making a scripted, linear campaign.

Our prototype design process ended up helping the situation. Starting from scratch, programmer Ike Ellis coded up the skeleton of the module that would become *Conquer the World*, which aimed to bring context and meaning to linked scenarios that change depending on choices the player makes throughout the game. This campaign mode went from a prototype experiment to a central selling point for the game in less than nine months, and we are planning a new version of *Conquer the World* for the next game.

5. Refusal to acknowledge the true state of the game in the final months. Our insistence that we could power through our bug counts at a faster-than-light clip meant that we worked our team much harder than we should have going into the final months of the project. There were a couple of modules that were going to cause us to slip by the month that we did, and recognizing reality sooner would have saved much of the team a death march that was extremely difficult for all involved.

On the next project, we’ll retain our faith in our programming team while making more effort to be up-front with ourselves about the status of the project at any given time. Specifically, we need to pay more attention to our bug counts as reflective of the state of the project. We will also be more careful about not calling something “done” until it is truly “done-done-done,” and adjust our schedules and expectations accordingly.

It Takes More Than Effort

We’re extremely happy with how *RISE OF NATIONS* has turned out. We started the company with a vision of how games should be created and how teams function best. In the end, the only tangible validation to our approach is the quality of our games. We said many times during the project that the gaming public only rewards success, not effort; we hope that *Rise of Nations* demonstrates our commitment to both. 🎮



MMORPGs:

for *Jive Magazine*, author Jewels writes about Iggy, whose addiction led to his disappearance into EVERQUEST and subsequent loss of his job and friends. Iggy emerged over a year later only when the server went down for maintenance one night. These are but a few of many similar stories.

Is MMORPG addiction real? The testimonials are compelling but they don't answer the fundamental question. To determine whether MMORPG addiction exists, the camps need to (a) agree on a workable definition

The massively multi-player online role-playing game (MMORPG) is the game industry's latest

progeny to spark volatile and polarized public debate. Supporters tout the genre as a great social advancement. Critics cite fans' overenthusiasm as indicative of a growing addiction problem. But so far, no one has proved conclusively whether MMORPG addiction is real.

The MMORPG allure. For hardcore gamers, the MMORPG attraction is manifest. They crave immersive story lines and environments, challenging levels, replayability, intelligent NPCs, intuitive controls, and inviting game design. Replayability is hard-wired into the MMORPG's persistent-world infrastructure (EVERQUEST's title is itself a replayability tautology). The allure for game makers includes charging customers' credit cards monthly fees for long-term revenue potential that goes far beyond shrinkwrapped software's point of sale.

The perceived addiction problem. For casual players and nongamers, the allure of MMORPGs is confusing and even frightening. Critics cite the same immersiveness and replayability as addiction triggers. Some critics (including some self-proclaimed MMORPG addicts) argue that MMORPGs have the potential to, and in fact have caused people to forget their real-world problems and responsibilities. In an article by author Jeffrey Russel Stark in *Self-Psychology Discussion Forum*, a player with the handle "Grezkul" asserts that a high school semester was "destroyed by an INSANE use of video games." In an article

Perfect Game or Dangerous Addiction?

of what constitutes such addiction, and (b) define the nature and extent of the problem. Nick Yee, an independent researcher and psychology degree candidate, conducted extensive research into the perceived MMORPG addiction problem. For purposes of his study, Yee defines addiction as "a recurring behavior that is unhealthy or self-destructive which the individual has difficulty ending" and concludes that "MMORPG addiction is a very real phenomenon...." Some psychologists support Yee's conclusion, but ultimately the American Psychiatric Association and/or the American Medical Association must weigh in on whether MMORPG addiction is a diagnosable mental disorder.

The public response. One public response to MMORPG excess has been the formation of various online support groups. EVERQUEST Widows is the best-known online discussion forum, its stated mission to "support each other, and to discuss the trials and tribulations of living in Real Life while our partner is immersed in EVERQUEST." Other support groups include Online Gamers Anonymous (www.olganon.org), DARK AGE OF CAMELOT Addiction (www.darkageworld.com/disc7_welc.htm), and Spouses Against EVERQUEST (<http://groups.yahoo.com/group/spousesagainsteverquest>).

Another public response has been threatened litigation. Jack

continued on page 47

continued from page 48

Thompson, a Miami attorney and outspoken critic of the game industry, announced his intention to sue Sony Online Entertainment on behalf of Elizabeth Wooley, whose son Shawn committed suicide in 2001 shortly after logging off of EVERQUEST. Thompson argues that SOE “knows [EVERQUEST] is an addictive game” and says he wants to “whack [SOE] with a verdict significantly large so that [MMORPG makers], out of fiscal self-interest, will put warning labels on ...” However, warning labels are an incomplete solution and likely would not dissuade hardcore MMORPG players from indulging in excess.

The case for self-regulation. The game industry must resolve the apparent MMORPG problem before state, local, and/or federal governments step in and set their own limits. State and local governments already have demonstrated their willingness to impose legislative restrictions by targeting the sale of “M”-rated games to minors. The game industry could try to preempt government intervention by investigating the MMORPG issue itself and implementing self-regulatory measures (as it already has done with the ESRB ratings system).

Once the game industry has a handle on the nature and extent of the problem, it can work to implement remedial measures. Taking a cue from the gambling industry, the companies that maintain MMORPGs and servers could use login data to identify excessive players and dis-

seminate information to help those players identify whether their excess is a problem and whether and where they can get help (provided such practices are addressed in the relevant end-user license agreements and privacy policies).

Depending upon the official classification of excess MMORPG playing as an addiction or health concern, such practices also may implicate state or federal regulations pertaining to the online collection and transmission of health care-related data. Developers also could write on-screen timers into the client software to help excessive players keep track of how long they have been logged in (some would-be addicts simply lose track of time). Ultimately, the solution boils down to the game industry identifying who, if anyone, might need helpful information, and then making that information available so that players can help themselves.

The debate over MMORPG addiction may continue for some time without any meaningful resolution. In the meantime the game industry must decide whether to own these issues and implement its own action plan, or allow litigants, legislatures, and courts to move forward in its stead. 

DAMON C. WATSON | *Damon is an attorney in the Entrepreneurial, Technology, and Commercial client service group at Bryan Cave LLP in Santa Monica, Calif. You can contact him at dcwatson@bryancave.com.*