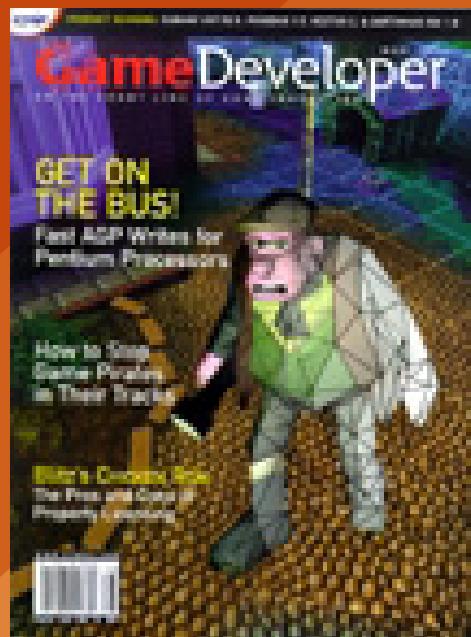


gd

GAME DEVELOPER MAGAZINE

MAY 2001





GAME PLAN

LETTER FROM THE EDITOR

Violence and Education

OK., so I've got to come clean on this whole violence thing. I don't like violent videogames. I don't like violent movies much either. My personal definition of violence includes gratuitous use of blood and gore. I don't care if the blood is green, and I don't care if they are zombies, aliens, or space marines — I'm just not interested in killing things or seeing things killed. But I respect that there are people out there who do enjoy violent themes in games and movies, and I don't want to limit their right to experience that. On the other hand, when the focus is on competition such as in COUNTER-STRIKE, I'm willing to wave away my concerns.

But does playing violent games make people violent? I certainly don't think so, not at our current stage of technology. However, every experience people have in life has the potential to teach them something. It's part of being human: we try to generalize every experience into a lesson or rule we can apply to life. So every experience you have during the day has the potential to educate.

One thing that a violent game can teach a person is how to be a sharpshooter. Those arcade games with the guns — they're great training devices for learning how to hit targets quickly and accurately. But does this mean that they're training you to be violent? This is a very fine distinction. If violent games really trained people to act violently, wouldn't John Carmack or Thresh have gone postal by now?

Given that games can teach people, why aren't there more fun educational games available? Do you remember what it was like to learn history, or physics? I remember my experience, sitting in an uncomfortable desk for hours at a time listening to an uninspired teacher drone on and on. As an industry, we could be making games which take the boredom out of school for the next generation of students. Can you imagine playing a QUAKE 3-quality game which has been designed specifically to teach you chemistry? Or art history?

I keep thinking about a good friend of mine who is an incredible DANCE DANCE

REVOLUTION (DDR) player. Now, I never would have imagined this fellow bopping around on a dance floor, but he recently won a Seattle-area DDR competition. As a result of his excitement about DDR, he's gotten into better shape, gained a large social circle, and improved his self-esteem. I doubt that DDR was specifically designed to produce these results, but it is a very pleasant side effect. Given that it is possible for games to affect people in this manner, what kind of games could we create with these potential results in mind?

Most of us think of games solely as a form of entertainment. But they are also a form of education. What is your game teaching your players?

This Month

I'm sure you've heard the statistics before: current top-selling PC games don't even utilize the bandwidth of the standard AGP bus, let alone get anywhere near AGP 4x performance. This month, Dean Macri of Intel shares some techniques you can use to optimize your AGP bus utilization.

Gavin Dodd from Insomniac Games spent many cycles working out a unique crack protection scheme for Insomniac's title SPYRO: YEAR OF THE DRAGON. His results? Preventing a fully cracked version of SPYRO from spreading around the Internet for two months. If you've published CD-based titles, you know that this is quite an accomplishment. Gavin shares his techniques this month with the hope that you too will be able to benefit from his research.

The movie *Chicken Run* was an animated sensation. The game CHICKEN RUN was one of the most elaborate cross-platform projects yet, appearing on Dreamcast, PlayStation, PC, and Game Boy Color. Dave Flynn and Dave Manuel from Blitz Games share their experiences building a game on this unique license in our Postmortem.

We hope you enjoy this month's *Game Developer*!

Game Developer

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

Publisher

Jennifer Pahlka jpahlka@cmp.com

EDITORIAL

Editor-In-Chief

Mark DeLoura mdejoura@cmp.com

Senior Editor

Jennifer Olsen jolsen@cmp.com

Managing Editor

Laura Huber lhuber@cmp.com

Production Editor

R.D.T. Byrd tbyrd@cmp.com

Editor-At-Large

Chris Hecker checker@d6.com

Contributing Editors

Daniel Huebner dan@gamasutra.com

Jeff Lander jeff@darwin3d.com

Mark Peasley mpeasley@gaspowered.com

Advisory Board

Hal Barwood LucasArts

Noah Falstein The Conspiracy

Brian Hook Independent

Susan Lee-Merrow Lucas Learning

Mark Miller Group Process Consulting

Paul Steed WildTangent

Dan Teven Teven Consulting

Rob Wyatt The Groove Alliance

ADVERTISING SALES

Director of Sales & Marketing

Greg Kerwin gkerwin@cmp.com t: 415.947.6218

National Sales Manager & Western Region, Silicon Valley & Asia

Jennifer Orvik jorvik@cmp.com t: 415.947.6217

Senior Account Manager, Eastern Region & Europe

Afton Thatcher athatcher@cmp.com t: 415.947.6224

Account Manager, Recruitment

Morgan Browning mbrowning@cmp.com t: 415.947.6225

Account Manager, Northern California

Susan Kirby skirby@cmp.com t: 415.947.6226

Sales Associate

Aaron Murawski amurawski@cmp.com t: 415.947.6227

ADVERTISING PRODUCTION

Senior Vice President/Production

Andrew A. Mickus

Advertising Production Coordinator

Kevin Chanel

Reprints

Stella Valdez t: 916.983.6971

GAMANETWORK MARKETING

Senior MarCom Manager

Jennifer McLean

Strategic Marketing Manager

Darrielle Ruff

Marketing Coordinator

Scott Lyon

Audience Development Coordinator

Jessica Shultz

Sales Marketing Associate

Jennifer Cereghetti



CIRCULATION

Group Circulation Director

Kathy Henry

Director of Audience Development

Henry Fung

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

Corporate President/COO

John Russell

CFO

John Day

Group President, Business Technology Group

Adam K. Marder

Group President, Specialized Technologies Group

Regina Starr Ridley

Group President, Channel Group

Pam Watkins

Group President, Electronics Group

Steve Weitzner

Senior Vice President, Human Resources

Leah Landro

Senior Vice President, Global Sales & Marketing

Bill Howard

Senior Vice President, Business Development

Vittoria Borazio

General Counsel

Sandra Grayson

Vice President, Creative Technologies

Philip Chapnick

GamaNetwork

A DIVISION OF CMP MEDIA INC.

SAYS YOU



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

Author Stresses Story over Style

Inspired by the Game Plan column "Telling Stories" (February 2001), I decided to write about an incident that contributes to the discussion of games as an art form discussion.

I'm a game designer at Satama in Tampere, Finland. In the last 20 years, I've witnessed the uprise of ever more visually stunning graphics engines and the downfall of story — I don't necessarily mean an absence of story, but badly written dialogue, lame characterization, and a lack of literary values. There is a complex and twisted story line in FINAL FANTASY VII, but the dialogue is appalling.

My mission as a game designer has been twofold: to write an adventure game in Finnish for the Finnish-speaking audience and to write it in a rich and elaborate way. For 20 months our team of four artists (myself, a graphic artist, an animator, and a sound designer) forged a very non-state-of-the-art, non-real-time-3D, non-force-feedback, sprite-based, top-down-view role-playing adventure game in the tradition of Super Nintendo's ZELDA. I don't want to stress the low-tech aspect of the game — it's not the thing that made this game any good. We wish we could have made this game in glorious real-time-3D, but we did not have that know-how or resources at our disposal. The game was written in Lingo (on Director), as that was

the programming language I was fluent in.

Only a few weeks after our finished game, GALILEI 2: ISLAND OF ADVENTURE, went public, the Finnish Ministry of Education awarded us with the annual Suomi award, given for a distinguished career in arts, a notable artistic achievement, or a promising breakthrough. It was a great honor to be awarded, not so much for the personal glory, but as an acknowledgement for the whole industry. It was the first ever cultural award for a game in Finland. Personally, I've always seen games as an art form, but here was recognition clear and strong that they could be elevated to that status in the public eye. It has been heartwarming to see that reviewers and the game players themselves have noted and valued the game's emphasis on storytelling and clever dialogue. From the feedback we've received, I've noted another interesting aspect: a majority of our game's players are women. Apparently you don't have to make "games for girls," just a game that's intellectual and nonviolent. The rest will follow.

Thank you for your attention. I hope this was good news to you too. I just had to use this opportunity to tell about this achievement as there has been so much talk about story and whether games will ever be an art form of their own.

Henri Roth
Satama Interactive
via e-mail

Reader Questions NPR

I'm a master's student at Johns Hopkins University, doing my thesis in the area of real-time non-photorealistic rendering (NPR). Since research in NPR is still new, it's difficult for me to evaluate the potential and usefulness of NPR in games, especially techniques such as painterly, pen-and-ink, and watercolor, although cartoon rendering does seem to be gaining popularity. I read Jeff Lander's article "Graphics Programming and the Tower of Babel" (Graphic Content, March 2001), and I had some questions. Are these "other" NPR techniques being used in games? I find it disappointing that "interactive frame rates" usually means a few frames per second. Jeff described methods of using Direct3D's new vertex shaders as a means of achieving cartoonlike shading. Can this be extended towards other techniques? Since most NPR techniques are basically texture-mapping techniques, I don't know if having this type of hardware support will help much. I'm very interested in real-time graphics in games and would very much like to contribute something in this new and seemingly hot area.

David Ko

The Johns Hopkins University
via e-mail

Send e-mail to editors@gdmag.com, or write to Game Developer, 600 Harrison St., San Francisco, CA 94107

Kludge by Tiger Byrd and Daniel Huebner



INDUSTRY WATCH

daniel huebner | THE BUZZ ABOUT THE GAME BIZ



Sega restructuring.

Sega president Isao Okawa has helped to ease some of his company's recent financial pain by giving his stake in Sega back to the company as a gift. Okawa gave up his 12.5 percent stake in Sega on January 31, totalling 19.87 million shares

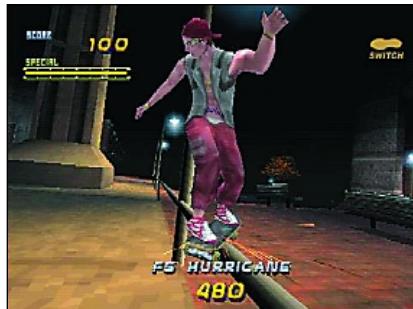
worth more than \$280 million at the time of the transaction. Okawa also gave up stakes in compa-

nies associated with Sega, including a 4.8 percent share of CSK Corp. and a 3 percent stake in ASCII Corp., for a total stock gift to Sega worth \$566.3 million.

Sega is looking to save costs by making job cuts and will accept voluntary retirement from 300 employees as part of a downsizing plan related to its departure from the console business. The company said it will post a \$11.2 million special loss for these retirement allowances but adds that in the long run the retirements will result in \$16.4 million in annual savings.

Sega's decision to exit the hardware market has also translated to layoffs for Sega of America. The company has cut jobs in departments tied to the Dreamcast console, largely in the marketing and quality assurance areas.

Positive financial forecast. Things are looking up for a number of the industry's biggest players. Activision reported income of \$20.5 million on revenues of \$264.5 million in the third quarter, slightly down from last year's third-quarter result of \$22.3 million in income on \$268.9 million in revenue. Meanwhile, mergers and reorganization are starting to pay dividends for Infogrames. The company's second-quarter revenues increased to \$127.5 million from last year's second-quarter revenues of \$109.7 million. Changes in income were impressive, as the company exchanged last year's second-quarter loss of \$118 million for this year's net income of \$16.5 million for the same period. Infogrames' second-quarter results include its October 2 merger with Infogrames North America. For its part, THQ announced earnings results for its fiscal fourth quarter which show net income improving 45 percent to a record \$21.5



Activision's TONY HAWK 2 (left), Infogrames' UNREAL TOURNAMENT (top right), and Ubi Soft's RAYMAN (bottom right). All three companies have posted impressive earnings despite an overall slump in the market.

million. Revenue for the fourth quarter also broke a company record, increasing 50 percent to \$190.9 million. Ubi Soft's fiscal third-quarter results show consolidated sales of \$100.7 million, an improvement of nearly 50 percent from last year.

Negative financials. Midway released its results for the fiscal second quarter, posting lower-than-expected earnings. The Chicago-based company lost \$3 million on revenue of \$76.9 million, down a whopping 48 percent from the \$147.6 million in revenue for the same period last year. 3DO is also feeling the pinch, revealing a fiscal third-quarter loss of \$12.3 million on revenues of \$29.9 million, compared to a profit of \$1.4 million on revenues of \$41.2 million for last year's third quarter. Both companies cited the current slump in game sales as the cause for their losses and expect larger returns in the coming quarters.

Game makers file suit against online pirates. Twelve game companies are bringing civil lawsuits against the operators of web sites alleged to be offering illegal game downloads. The game companies (Activision, Capcom, Eidos, EA, Havas, LucasArts, Interplay, Midway, Microsoft, 3DO, Sierra, and Nintendo) allege that the defendants offered pirated copies of hundreds of titles on various "warez" sites on the Internet. The suits seek court assistance to shut down the offending sites and to bring monetary penalties that could range as high as \$150,000 for each copyright violated. "These cases target Internet warez and ROM sites where games can be downloaded," explained Doug Lowenstein, president of the Interactive Digital Software Association, the trade group which repre-

sents game publishers. "Those engaged in piracy need to understand that they will be targeted for legal action and that they will pay a price for their illegal conduct."

Ubi Soft acquires Blue Byte.

Ubi Soft has acquired German developer and publisher Blue Byte Software. Privately held Blue Byte, which has 64 employees, is the maker of the SETTLERS and BATTLE ISLE game series. The company was founded in 1988 by the current head of the company, Thomas

Hertzler, and has operations in Germany, the United States, and the United Kingdom. The acquisition was consolidated into the Ubi Soft accounts as of February 6, 2001, and Ubi Soft reports that it will have a positive impact on its results in 2001.

Blizzard files Diablo movie suit. Blizzard Entertainment has filed suit to prevent New Line Cinema from releasing a movie under the "Diablo" name. Blizzard contends that New Line is trying to use the popularity of the DIABLO videogame franchise to promote its film, which is unrelated to Blizzard's game of the same name. Furthermore, Blizzard has plans to produce a movie of its own based on the DIABLO franchise and using the DIABLO brand. Blizzard was granted a DIABLO movie trademark in 2000, though New Line contests its validity. New Line's *Diablo* film, which began production last December, features Vin Diesel and revolves around a drug lord known as Diablo. Blizzard is seeking an injunction to prevent the release or promotion of the film under the *Diablo* name. ⚡



UPCOMING EVENTS CALENDAR

MEDPI 2001 SOFTWARE

GRIMALDI FORUM
Monte Carlo, Monaco
June 26–29, 2001
Cost: variable
www.reedexpo.fr/anglais/factsheet.html?salon=28

PRODUCT REVIEWS



THE SKINNY ON NEW TOOLS



The Cubase VST/32 5 interface.

Cubase VST/32 5

by andrew boyd

Cubase VST/32 5 is German developer Steinberg's flagship entry into the important digital audio sequencer market. It offers competitive functionality (deep and robust MIDI editing alongside a high-resolution multi-track audio system) and some unique extras. Introduced in 1989, Cubase is certainly one of the established players in the segment. Steinberg's Virtual Studio Technology (VST) has become popular as a cross-platform foundation for digital audio, with countless compatible effects plug-ins and virtual instruments on offer from a number of developers. Cubase VST/32 5 claims an impressive feature set including audio resolutions up to 32-bit, Apogee UV22 dithering, four bands of parametric EQ, a dynamics processor, four insert effects, and eight aux sends per channel. It is available for PC and Mac; I looked at the PC version.

Now, there are certain problems with reviewing popular, mature, full-featured

products like Cubase. How does one delve deep enough into such a huge application to give meaningful advice to prospective buyers? How does one secure the credibility to critique an application used regularly by professionals for high-end work? How does one distinguish raw prejudice from thoughtful analysis? Here's how I solve these problems: Cubase VST/32 5 is a fine, well-made application capable of solving just about any MIDI or audio recording, editing, mixing, or processing task it's put to. It's extremely powerful, sounds great, and is rock stable. Whether you actually like it and find it useful is a mostly personal matter. Whatever you need, Cubase can do it, and do it very well. Go online (www.steinberg.net) and download the fully functional demo.

Cubase starts with a deficit — it uses a dongle for copy protection. I know, musicians are criminals, and without a brutish and despotic lockdown the application would be copied ruthlessly. This is exceed-

ingly unfortunate, but it's hard to deny that it's probably true. Otherwise, installation was smooth and quick. Lots of good electronic documentation installs along with the application, and so do a couple of useful demonstration and tutorial songs. A well-written Quick Start guide is included in the package, beginning with chapters called "What is MIDI?" and "What is Digital Audio?" and running through nicely organized examples of most major features of the software. Beginners are in good hands, and more experienced users moving over from other applications will find a handy reference to the particulars of Cubase.

I tested Cubase in Windows 2000 running on a 700MHz Pentium III with 256MB RAM and a 30GB hard drive, decent, if not cutting-edge, computer hardware. But I had trouble getting my pro-level audio hardware to work under Windows 2000, so I finally decided to run Cubase with a trusty Sound Blaster Live! doing MIDI and audio duty. This was lower-end than I would have preferred, but worked well. I was unable to assess Cubase's high-end audio features, but I certainly believe they work as advertised. I did several projects in Cubase: I edited the demo songs, imported a video and its SFX tracks in order to work on MIDI music tracks, and created a loop-based piece with lots of audio tracks. In each case, my results aligned well with my expectations. The MIDI power is formidable. As for audio, basic mixing processes sound great, the EQ and dynamics are clean and effective, and other included effects (reverbs and such) are more than acceptable, if less than spectacular. The included analog-modeling synthesizer is a nice touch and great fun. But then, who cares about included effects when there's an almost limitless supply of VST-compatible effects available? Cubase also supports DirectX format plug-ins, so third-party support will not be a problem.

The meat of the work you do in Cubase takes place in the Arrange window, and this window works very well indeed. Track names are aligned down the left side, their contents graphically across the right. MIDI and audio coexist happily here on the right side, and editing is easy with grid lines that can be turned on and



Cubase's VST channel settings.

off, snap-to values that are quickly changed, and easy color coding. Sliders in the lower-right corner adjust the overall horizontal and vertical zoom levels, and horizontal zoom can also be adjusted independently and arbitrarily per track. Expander buttons on the far left of the window open a section with functional details of the selected track. Dragging a file from Windows Explorer imports it as an event. Right-clicking on an event opens a toolbar menu, and double-clicking opens the event in a specialized editor. This is all intuitive, informative, and contextually appropriate.

And on the whole, using Cubase makes sense enough. It didn't take me long to feel competent — but I never really settled in and got comfortable. It's not the best-looking application for one thing (garish and busy to my eyes), and its operation is often illogical, even inconsistent. Sometimes a display window is a display window, sometimes it's an edit box, and sometimes it's a drop-down list. There's no consistent way to tell which. Why is the Create Track command in the Structure menu, while the Delete Track command is in the Edit menu? Why are windows called Panels some times and Windows other times, and why are some things that in most ways seem like Panels called Modules? There are probably reasonable explanations for all of these, but I was unable to discover them.

When a large number of windows are open at once, the screen can get very cluttered and confusing, and Cubase opens a new window for virtually every operation. For example, the mixer window has four LED-style indicators labeled INS, DYN, FX, and EQ. Clicking on any of the four indicators opens a channel-specific window called the VST Channel Settings win-

dow. This is like a "fat channel," or an exploded view, dedicated to the operation of a single channel. Here, an apparently identical set of the indicators appears again, but now clicking on them generates different results. For instance, clicking on the INS and FX indicators here has no effect. But immediately to their right are two columns of buttons and knobs and drop-down lists labeled Inserts and Sends. In the Inserts column, clicking on an effect name drop-down list opens a window where you choose your desired insert effect. Clicking the On button illuminates the INS indicators in this window and on the mixer. To adjust the parameters of the effect, you click on the Edit button, which opens another window. And so on.

For me, the flexibility this represents is overshadowed by the resultant visual chaos and the constant danger of adjusting the wrong parameters. A fat channel display that always and only shows the currently selected channel would be more useful than a screen full of them, all looking almost exactly alike. The title bar of each VST Channel Settings window does contain the name of the channel it represents, but it's easy to imagine naming two channels the same thing — say, "guitar" — and then really screwing up the settings on one because both VST Channel Settings windows are open and it isn't clear upon which you are working.

I found it hard to create a workspace wherein I could easily glance at important information. The INS indicator in the mixer can show that one or more inserts are active on a channel but not how many or which ones. The VST Channel Settings window can show which and how many inserts are active and the effects assigned but not any of the effects' settings. The effects settings windows can show the settings for that particular effect but nothing else about the channel. In a large mix with dozens of channels, each with EQ, dynamics processing, and three or four insert effects, finding immediately relevant mix information becomes very frustrating.

Of course, a program like this has so much to communicate that there will always be particulars and idiosyncrasies in any scheme it uses. And it's possible that I missed things — it can take months to really get into an application like this.

Nonetheless, I believe the inconsistencies in the interface detract from Cubase's usability. Whether or not these are a problem for you is, well, for you to decide.

If you're a pro happily using one of the other digital audio sequencing packages, there may not be a compelling reason to switch to Cubase. If you're a beginner just getting into large-scale digital music production, Cubase absolutely has a lot to offer. It's so powerful that you're not likely to push its limits anytime soon, and its helpful documentation will speed you on your way. Once you are accustomed to its interface, you might learn to love it, and the support offered to its VST engine is phenomenal. If you're a seasoned pro with reason to switch applications, Cubase is worth a look. It's got the quality to run with anything, and its depth of features is outstanding.

CUBASE VST/32 5



STATS

STEINBERG MEDIA TECHNOLOGIES AG

Eiffestr. 596

20537 Hamburg, Germany

www.steinberg.net

PRICE

\$799

SYSTEM REQUIREMENTS

PC: 200MHz Pentium, 64MB RAM,

Windows 95/98/2000; Mac: Power PC

200MHz, 64MB RAM, MAC OS 8.5 or

greater

PROS

1. Incredibly deep and powerful MIDI and audio functionality runs with the best.
2. VST architecture guarantees huge palette of after-market goodies.
3. Arrange window is a powerful way to envision MIDI and audio data.

CONS

1. Odd inconsistencies in interface slow down workflow.
2. Bright and garish visuals cause fatigue, confusion.
3. Dongle copy protection.



THE CHAOS GROUP'S PHOENIX 1.5

by steven thompson

Creating realistic fire effects in 3D has always been a challenge. The Chaos Group stepped up to this challenge by creating Phoenix, a 3DS Max plug-in distributed by Digimation.

Phoenix produces realistic, natural fire effects that react to the movements and modifications of their emitters. A single candle flame, a campfire, or a raging fireball can be generated quickly and easily while giving the user an astounding amount of control over the fire's color, shape, movement, and interaction with objects in the scene. The latest version, Phoenix 1.5, adds some powerful new features to its already sophisticated toolset. Phoenix Lights, Phoenix Render Effect, and Phoenix Fractal Map give users more editing power and reduce production time. The new version boasts improvements in speed and stability, better interactive controls, and greater flexibility for emitters and deflector objects.

The most significant addition to Phoenix 1.5 is the Phoenix Light. Adding a single Phoenix Light to a scene creates a series of lights for each flame object, or Shapenode, within a Phoenix atmospheric effect. The light can be attached to multiple atmospheres and can affect multiple emitters within each atmosphere. For example, let's say you have two Phoenix atmospherics in a Max scene, one with two orange-colored torch flames and another with two blue-colored torch flames. Adding one Phoenix Light to the scene and attaching it to both atmospherics enables all of the torch flames to emit light based on the color of the flames. Because Phoenix Lights derive their color from the parent flame, the intensity and color of the light emitted are based on a dynamic fractal, which automatically produces natural flickers from the flame source. Need to tweak the color of a Phoenix Light? No problem. A color-cor-

rection swatch enables you to add color correction to the light emitted from the flames. You can also choose whether the color is additive, subtractive, or a combination of these.

A Phoenix Light works by automatically placing a large number of lights along the splines that define the shape of the

flames. The number and intensity of lights added to the splines can be controlled, so you can balance render speed and quality. The huge advantage gained from Phoenix Lights is that users no longer have to create scores of animated lights to mimic the natural

light emitted from the fire effect. As you might expect, render times for scenes containing Phoenix Lights can be a bit long, particularly if cast shadows are active. This can be remedied simply by reducing the number of lights per spline and disabling the Cast Shadow parameter during test renders.

An important addition to Phoenix 1.5 is the Phoenix Render Effect. The Render Effect copies Phoenix render information into the appropriate G-buffer channel for use in postproduction or with other Max plug-ins such as Lens Effects. Material Effects ID, G-buffer ID, non-clamped color, Z-buffer, RGB color, UV, and coverage channels are supported. Each channel also has a number of common controls that can be flagged, such as brightness and/or density, raytrace output (RT), and channel clearing (CLR).

Want to make a fire that won't take a long time to render? Enter the Phoenix Fractal Map. This new feature takes Phoenix's Color Fractal and routes it to a procedural map that can be applied to any material mapping channel in Max. The purpose of this feature is to create quick, relatively distant, or small fires procedurally, without having to create a processor-intensive 3D volumetric fire.

If you're considering upgrading from Phoenix 1.0, the Phoenix Light feature



A scene lit with a single Phoenix Light. Natural-looking flickers are emitted as the flame moves.

alone is well worth the \$50 upgrade cost. Tweaking the plethora of controls can be tedious at times, but this comes with the territory when you have so many editing options. Phoenix 1.5 is available for 3DS Max 3 at a list price of \$395. Digimation promises registered Phoenix users a free upgrade for 3DS Max 4 when available.

| **PHOENIX 1.5**
The Chaos Group (distributed by Digimation)
www.digimation.com

CODEPLAY'S VECTORC

by brian sharp

Codeplay promotes VectorC as "a standard C compiler that can automatically create highly optimized code for PCs that make use of MMX, AMD 3DNow! and Intel Streaming SIMD extensions." It supports the specific instruction sets of a large variety of processors, including the Pentium III, Pentium 4, and Athlon. Its target market is development where optimization is essential, and games fall neatly into that domain.

VectorC is specifically designed to vectorize generated code — to make use of SIMD-style instructions on processors — which few other standard compilers purport to do as effectively. For game developers who need a few more percentage points out of their code, VectorC could be a godsend if it performs as promised.

When I first read over the information on Codeplay's web site, I had some apprehensions. The lack of a troubleshooting section in the documentation was a bit concerning. As it turned out, it wasn't necessary. Installation of the compiler was trivial, and configuring it to run within Microsoft's Visual C++ was also a snap. Choosing compiler options was simple enough, thanks to the great documentation, which is written in a very straightforward, readable style.

I started out with a C++ application and pulled hotspots out as C code and used the VectorC compiler on them. Unfortunately, the act of porting small sections of my C++ code to C proved harder than I'd anticipated, which leads me to my main concern about VectorC: if you write in C++, it may be tricky to isolate sections of your code after you've written them and



apply this compiler to them. However, Codeplay claims that work on C++ support is under way, so it may be that my concern is short-lived.

I then decided that the most efficient way to test VectorC's capabilities would be with an existing, computationally intensive application that was already pure C. I downloaded the source code to QUAKE 1 (<ftp://ftp.idsoftware.com/idstuff/source/q1source.zip>) and built it using the pure C versions of functions — none of the handwritten assembly — with relative ease under Visual C++ 6. In release mode, it ran fairly correctly, with only a few minor rendering artifacts.

The real test, then, was to build QUAKE 1 with VectorC. I was impressed: it generated four errors, but other than that, it built correctly. It took five minutes to move from the Visual C++ compiler to the VectorC compiler and address all the resulting issues. While QUAKE had some artifacts at high optimization levels, at lower levels it ran correctly on the first shot.

My end test was to pit my Visual C++ 6 executable against my VectorC executable built at optimization level 3 (on a scale from 0 to 10, 10 being the most robustly optimized). I recorded a quick demo. The results were 59.7 frames per second from the Visual C++ executable, to 63.3 frames per second from the VectorC executable, a not-insignificant speed gain from a relatively low optimization level. I was pleased with how quickly I was able to produce good results.

A potential drawback of VectorC is that the Professional Edition is rather expensive at \$750. However, there is a Special Edition (which lacks Pentium 4 and Athlon optimizations) available for only \$80. Overall, VectorC is a novel and promising compiler. If your game could stand to be faster, it's almost certainly worth trying out the Special Edition of VectorC and perhaps then upgrading if your results are good. Games written entirely in heavily object-oriented C++ might not be able to benefit quite as easily from VectorC, but with a C++ version of the compiler in the works, even that problem will hopefully be resolved soon.

| VECTORC | Codeplay | www.codeplay.com

AVID'S SOFTIMAGE XSI 1.5

by *david stripinis*

Softimage is back. That's my reaction to Softimage XSI 1.5. Rich with new features and refinements to existing ones, this is a major release of Avid's 3D production tool. My only logical assumption is that all these features were originally intended for version 1.0, before market forces hastened its release.

Most significant in this release is the addition of a complete suite of polygon and subdivision surface tools. XSI 1.0 had virtually no polygonal modeling support, short of simple primitives and imported models. This has been rectified in a most impressive manner. Artists have the ability to modify existing geometry or build it vertex by vertex. Whenever a component object (an edge, point, or polygonal face) is selected, right-clicking will bring up a context-sensitive set of operations, speeding up modeling quite a bit. The toolset seems very similar to those offered by Winged Edge Technologies' products, including powerful beveling, extrusion, and polygon-bridging. XSI is not restricted to closed shells, however, which is of great benefit to game artists trying to optimize every single polygon of a model.

Refinement of detail within a model is a breeze, with an impressive edge-splitting tool and multiple methods of subdividing components. One fault was the lack of any kind of automated symmetry tool, which enhances the modeling of polygonal characters. This addition, as well as a way to select edge loops, would put XSI in a position to honestly claim the role as the premier polygonal modeling tool available to artists today.

All these pretty polygons would be useless without textures, and XSI's ability to apply texture coordinates to polygonal models is nearly as powerful as its facility at creating the models themselves. Many of the same hotkeys and transformation tools work in the texture editor, although I was quite disheartened to see the powerful proportional modification tool did not. XSI also handles multiple UV channels, which some next-generation consoles and PC products support.

The other major modeling addition is subdivision surfaces. Support for both Catmull-Clark and Doo-Sabin surfaces are

included and can be changed via the Property Editor at any point. Subdivision surfaces are created by selecting a polygonal object and pressing a button. XSI retains a connection between the two objects, so editing the original alters the resulting surface. XSI also provides fully variable creasing of both vertices and edges. One major pitfall is the apparent inability to work with a wireframe control object and a shaded resultant surface.

Other additions to the modeling functions of XSI include the capability to do more precise modeling, with the addition of snapping tools and reference. Many of the NURBS tools have been expanded, particularly in the area of curve editing. Incidentally, NURBS geometry can also be converted to polygons.

One interesting addition is the head generator. Along with bipedal skeletons of varying complexity and full-body models of both male and female forms, there are simple polygonal heads. The quality is that of a stock model collection but with the capability to edit parameters such as the cheekbones or the muscle tension of the neck. While not suitable for a main character, it does provide a simple and adequate method for adding variety to background characters.

After polygons, the most glaring absence from version 1.0 was a dopesheet. A dopesheet allows easy access to keyframes, and thankfully, Avid has included one in this release. Also included is a spreadsheet providing access to channel data for any number of objects at the same time.

One other nice addition to the interface is the ability to launch a web browser within a viewport. Being able to have a web tutorial available without toggling between applications is a really nice feature. In fact, in everything from the new tutorial book, which lies flat even while open, to the video walkthroughs of features, Avid has gone out of its way to make XSI approachable to users of every level.

With powerful polygonal tools, strong support for subdivision surfaces, enhancements to the user interface, and user-inspired tweaks, this is a worthy successor to the heritage of the Softimage product line. Softimage is back, and I couldn't be happier.

| SOFTIMAGE XSI 1.5 | Avid | www.softimage.com

Dave Aronson Directing Direct3D

After working with game developers for years at Nichimen Graphics, Dave Aronson recently stepped into the role of program manager for Direct3D and OpenGL at Microsoft. We caught up with him in Seattle to talk about the future of these two APIs.

Game Developer. So, what do you do at Microsoft?

Dave Aronson. I'm a program manager in the D3DX team, doing work with content creation tool companies and input from our graphics advisory board. My background has been dealing with customers and trying to make that pipeline work. We're finding that users' expectations for OpenGL are much different from those for Direct3D, which is kind of shocking, but there are different levels of expectations and desires.

GD. In the game community, there is a perception that the OpenGL Architectural Review Board moves at a glacial pace, and people are frustrated that new graphics features aren't being incorporated into the core OpenGL quickly enough.

DA. I'd say probably the majority of people who use OpenGL aren't game people. I'm just pulling these numbers out of the air, but I would guess 90 percent of the Direct3D customers are game companies. With OpenGL, it's probably more like 30 percent.

GD. Are you worried at all about cross-platform compatibility or are you just concentrating on Win32?

DA. On the Direct3D side or the OpenGL side? On the Direct3D side there are some issues with it being cross-platform. As we move forward into things like .NET, things change. For cross-platform titles people use OpenGL, and that's one of the major reasons why people use OpenGL versus Direct3D. It's our concern to make the Windows platform experience better, but as Microsoft's business grows outside of the Windows platform, anything is a possibility.

GD. Which game platforms have you worked on?

DA. I did 3DO for about a week. Part of the Game Exchange 1 libraries would take data and convert to Saturn, N64, Playstation, 3DO, or PC. So you could have the same animation playing on all of them. I did a little Playstation 2 work before I left Nichimen — right now I'm not doing anything with Playstation 2 or Gamecube, but I still talk to people who are doing that stuff. I don't think cross-platform game development is going to go away right now.

GD. It's more important now than ever. With so many platforms, you have to do multiple SKUs in order to recoup your costs.

DA. It's difficult moneywise and development-time-wise. Games take longer and longer to create. You can no longer sit in your garage, just two of you cranking out a game, one guy doing art and one guy doing programming. I'm sure that games are going to be coming out on every platform, but it will require significant work — it'll be interesting to see what types of trade-offs people make. Obviously PC and Xbox games will be an easier transition than going from PC to PS2 or Gamecube.

GD. There's so much built into Direct3D now, things like N-meshes and progressive meshes, that if you do a PC title with

Direct3D, and you want to port to a console besides Xbox, it's going to be trouble.

DA. I think you will end up having lots of differences between systems, where some systems can paint a lot of polygons really fast so they can do more polygons, whereas some systems can do cooler things with each polygon. I vote for the systems that can do cooler things with the polygons versus more polygons in general. It's kind of like the multi-texturing trade-off that happened a few years ago, you can either precombine the textures or just have lots of textures going on at the same time.

GD. You mentioned D3DX. What's that?

DA. I don't think a lot of people have known about D3DX, but it's something that makes a huge difference between Direct3D and other APIs. Basically it's a set of utilities that help the gaming experience, like single-skinning functions, shader helpers, font utilities, and so on. There haven't been any other toolsets that I know of that partner along with the API. D3DX will be a huge factor in making PC games look awesome.

GD. All of Direct3D and D3DX are used for Xbox, right?

DA. I can't comment too much on what they have specifically, but they certainly started off with Direct3D and they have access to the D3DX stuff, so hopefully what they choose is pretty close to what we have, because we think that it's pretty clean.

GD. But you guys basically split the tree?

DA. Basically we work with them, and hopefully Xbox will be in the DirectX space, so a lot of the same stuff could be used. Whether or not they choose to use it is up to them.

GD. Is it hard to support both Direct3D and OpenGL?

DA. In my mind I see it as pretty clear, because they're different markets. Game developers involved with OpenGL have work to do if they really want it to be a game API. The hardware and software vendors that are traditionally in the OpenGL market can't really do what they need to do using Direct3D right now. As Direct3D gets more features, those delineations are disappearing. I think you're probably looking at very few things now that Direct3D can't do, and a lot more things that Direct3D can do on top of OpenGL.

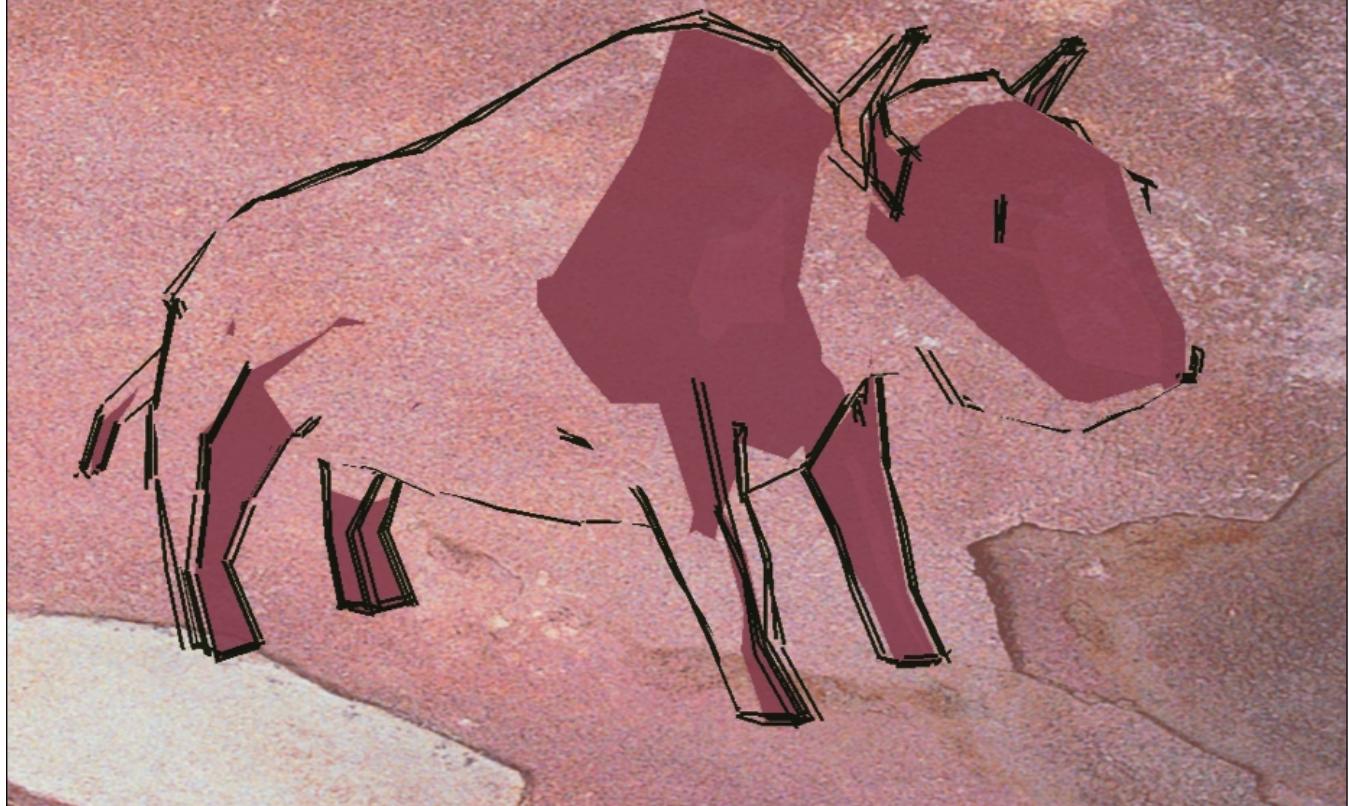
GD. Where do you think graphics are going from here? Graphics hardware is getting very advanced.

DA. You're going to see more programmability — but it's going to be the same phenomenon as with console games. In the first year, the games are pretty good but they're only a little different from what came before. Then developers figure it out and do some amazing stuff. We'll see things that are completely programmable, then people will go, "It's not as good because they didn't use the traditional ways." So we need to find that happy medium. ☺



ABOVE. Microsoft's Dave Aronson.

Images from Deep in the Programmer's Cave



Long before the first human used mathematics to calculate the area of a circle or the speed of a falling rock, people recorded their experiences and feelings through artistic expression. Early man used the materials in his surroundings for utilitarian purposes as well as to express his developing humanity. History was recorded with ornate carvings on stone and wood. Elaborate paintings deep within caves told of both significant conquests and the frustrations of everyday life.

Like many programmers, I often find myself digging deep into my forebrain, trying to get in touch with this primitive artistic aptitude. I want to get beyond my rather utilitarian duties and express more basic artistic emotions. My artistic friends often claim I must have had that part of my brain

removed early in my life. At that point I grab my big caveman club and go do some artist-bashing of my own.

Still, the urge to create persists. This month, I am going to try to re-create images in the style of these early cave paintings. The dominant feature of this type of drawing is the charcoal sketch lines that define the outline of the object. We have run into this before in the cartoon rendering system that I described in this column last year ("Return to Cartoon Central," August 2000). When implementing the ink lines for

the cartoon system, I simply rendered all of the back-facing edges of the mesh and relied on the Z-buffer to sort the rendering out. That gave adequate though not particularly flexible or stylistic results. In order to create the more random, hand-drawn look, I will need to calculate the actual edges that make up the silhouette of the object.

Tracing the Outline

Let me recap what makes up a silhouette edge. A triangular mesh is obvi-



JEFF LANDER | When not trying to make fire in his cave on the hill, Jeff can be found trying to evolve into someone more artistic at Darwin 3D. Send him your evolutionary accomplishments at jeffl@darwin3d.com.

ously composed of triangles. Each triangle is made of three edges which connect the three vertices of the triangle. In a closed, regular mesh, each edge of the triangle is shared with another triangle. For an edge to be on the silhouette of the object, one of the triangles that share the edge must be facing the viewer and the other must be facing away. Once I have determined the two faces that share an edge, I can calculate whether it is on the silhouette easily by looking at the face normals.

So the first step is to create an edge database that contains all the edges in the mesh that could possibly make up the silhouette. I will need to store two indices into the model vertex list. These indices will point to the two endpoints of the edge. In addition, I want to store the normals for each of the two faces to which the edge is connected.

Given a triangular mesh, I can go through each triangle and generate this edge database. I want to make sure that each edge is only represented in the database once. That means I need to be aware that the order of vertices is most likely reversed in the second connected triangle.

Once I have collected all the edges into the database, I have the complete potential silhouette edge pool for the model. As you can see in the clock in Figure 1, there are quite a few edges in the model. In this particular image, there are 444 edges. It should be apparent, however, that many of them couldn't possibly be part of the silhouette. For example, some of the edges on the front face of the model are clearly never part of the silhouette, so I don't want to consider them. If the normals of the two triangles that share an edge are equal, the two triangles are planar, and that edge cannot be part of the silhouette.

I still may actually want to draw that edge in some cases. For example, if the edge makes up a boundary between two

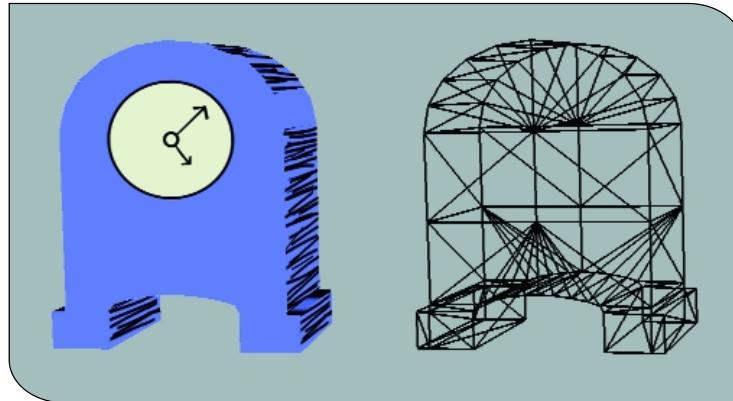
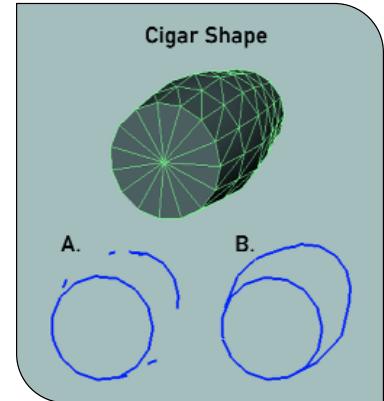


FIGURE 1 (left). A mesh clock and its edge database. **FIGURE 2 (right).** Slightly nonplanar edges rejected (A); Completely planar edges rejected only (B).

different textures or materials, I may want to always draw the edge if it is within view. However, most of the time, this is not the case.

So, I now have established my first rule that will reduce the number of edges I have under consideration. If the adjoining triangles are planar, the edge is rejected. I can test for this condition by seeing if the dot product between the two triangles is equal to one. If they are, the edge can be dumped. On the clock model, this simple test drops the edge count from 444 edges down to 256.

You might be tempted to reject the edge if the angle between the adjacent triangles is very small, say two or three degrees. That seems pretty close to planar. However, let me use an example to explain why that will not work. Let's assume we have a model of a cylinder with five divisions along its length. Now scale the vertices at the center of the cylinder so that they are slightly larger than the two ends. Scale the vertices at every other division along the cylinder so they smoothly interpolate to the two end caps. This should give you an object that is roughly cigar shaped. If the scaling is not great, the connected angle between triangles along the length will not be greater than two degrees. As you can see in Figure 2, the problem is that if you view the cylinder along its length, even this slight angle will create the need for a silhouette edge along the length. It's clear that even the slightest change in angle greater than zero degrees could potentially create the need for a silhouette. So it is



probably better to eliminate only completely planar edges.

To Draw or Not to Draw

Now that I have my database of edges, I need to decide what should be drawn for a given view. As I said earlier, I can use the edge-triangle normals to determine if an edge is on the silhouette.

I can find the angle between the viewer and the triangle face by using the dot product. The dot product of the triangle normal and the view vector is equal to the cosine of the angle between them. You may be tempted simply to use the Z-axis as the view vector. However, due to the perspective projection normally used in 3D view, this will yield visual problems as the object goes to the edges of the display. To counteract this problem, the view vector is usually defined as the vector from a point on the edge in question to the eye vector.

This gives me a couple of mathematical equations:

$$\text{dot}_1 = (N_1 \bullet (E - V))$$

$$\text{dot}_2 = (N_2 \bullet (E - V))$$

where N_1 and N_2 are the shared triangle normals, E is the eye point, and V is a point on the edge in question.

As a trivial test, I know that I probably always want to draw "hard" edges if they are facing the view. A hard edge is an edge where the angle between the adjacent triangle, or "crease angle," is greater than a defined amount. I usually find that 60 degrees works pretty well for defining hard

edges; however, this value can be set for each model. For nondeforming models, this test can also be done ahead of time and the edge flagged if it is a hard edge.

This gives me three tests to determine if I should draw an edge. I should do so if:

- $dot_1 > 0$ and $dot_2 < 0$ (the edge is on the silhouette),
- $dot_2 > 0$ and $dot_1 < 0$ (the edge is also on the silhouette), or
- The edge is “hard” and $dot_1 < 0$ or $dot_2 < 0$ (at least one of the connected triangles is facing forward).

That will render the set of edges which are hard or make up the silhouette. The next step is to give the lines a form of random, sketched feel.

Wiggle It Just a Little Bit

To create the look of someone sketching in a more natural manner, I want to vary the lines a bit. The first step to creating more natural-looking lines is to draw the edges multiple times, randomly offsetting the end positions slightly each time. Just this simple step helps a great deal. However, it also adds another problem. Since each edge is considered separately, these random offsets are applied to each edge. The result is a slightly random but very chaotic effect, as each edge may no longer be connected to the next one. That is not generally how people sketch objects, though it creates a pretty interesting look that may be useful in some applications.

In order to re-create a look more similar to human sketches, I need to consider more than the single edge. I need to look for a long series of edges combined to create a larger stroke that I can draw all at once.

To create organized strokes from a set of unorganized edge data is going to take some data shuffling. I could look for connected edges and start drawing them together; however, that method would create unorganized strokes that wander all over the model. The look I am trying to achieve is much more organized. An artist would probably not sketch lines that start at random and wander.

In order to organize the strokes in some manner, I made some assumptions. My virtual artist will tend to draw strokes

from top to bottom and from left to right. That seems a valid assumption given the western style of writing. So, the first step is to organize all the edges in this manner. If the dominant direction of an edge is top to bottom, I will then organize the index pointers so the vertex indices go top to bottom. Likewise, if the dominant edge direction is side to side, I will then organize the indices left to right.

Now I begin the automatic search for strokes — you can think of each edge as having a head and a tail which correspond to the first and second vertex index. I start at any point in the edge database and look at an edge. I then attempt to move backward up the stroke to find a potential starting point. This is accomplished by searching the edge database for an edge with a tail index value that is equal to the head index value of my current edge. If I find the edge that is ahead of the current one, I need to make sure it should be part of the same stroke.

The dot product is used once again to see if a candidate edge should be part of the same stroke. The test determines the angles between the two edges. If the angle between them is less than a given value (I have found that 36 degrees works well), it should be considered part of the current stroke. At this point, the search for the head of the stroke continues until either no other edges can be found or the angle between them is too severe. The current edge is then marked as the head of the current stroke.

Once the head of a stroke is determined, the system works its way down the stroke looking for the end using the same method of finding the next edge and seeing if it qualifies. Once this is finished, I have a complete stroke. I should note that this routine could be easily modified to limit the maximum stroke length to a defined value. For example, a certain artistic style may limit the length of a sin-

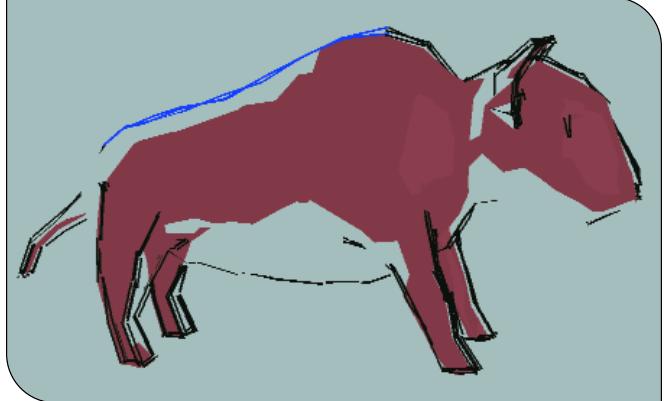


FIGURE 3. The stroke on the back of the buffalo.

gle stroke to a few edge segments. My current implementation allows strokes of arbitrary length as long as the stroke steps obey the angle rules.

Stroke results for meshes vary greatly depending on the mesh layout. However, fairly long strokes of more than eight edges are typically achieved. In Figure 3, you can see a typical model with a stroke that has been identified along the back of the buffalo object.

Stroke Man, Stroke Man

Once I have built my list of possible strokes from the edge database, I can start drawing. Now I can use the random jitter that I discussed earlier on each vertex along the length of a stroke. This will give the random look to each pass but still keep the strokes as one continuous movement. I can compute a new random jitter each frame or precompute it and store it in the edge structure. Both methods have interesting properties. If I compute the jitter randomly each frame, the strokes will look different every frame. While this creates an interesting “constantly being drawn” type of effect, there is a real problem with frame-to-frame coherence. The constant movement is a bit distracting. If the jitter is precomputed, the model looks correct as it moves and turns; however, the random effect is less noticeable. Since it is easy to do, it makes sense to have both methods available.

I should say another word about frame-to-frame coherence. This is an important issue with regards to artistic rendering techniques. It is quite easy to come up

with a technique that creates a very interesting style when viewing a static scene. However, when the model or the view changes, the effect can be overwhelmed by the frame-to-frame changes. It is important to deal with this issue when working on a technique.

When researching this idea, I looked at a couple of papers that calculated the stroke lines by doing image processing computations on the 2D rendered frame buffer (see *For More Information*). This method limits potential overdraw issues and allows for longer stroke lengths that don't follow the actual topology of the model. However, frame-to-frame artifacts are particularly noticeable using this technique.

Rendering the strokes allows for a great deal of opportunity to create different forms of effects. The most direct and simple method is to draw the strokes with simple line primitives. I found that it is interesting to change the color and the transparency value for the line at various points along the stroke. This gives a more natural look to the drawn line.

Another improvement is to go beyond simple line primitives and actually draw polygon primitives along the length of the stroke. Using this method, it's possible to vary the stroke width along its length. The stroke polygons are created by building a triangle strip whose vertices are offset from the mesh vertices in the plane of the screen. This displacement vector can be determined by taking the cross product of the edge and the view vector. This is very similar to the billboard method used to create textured sprites that always face the viewer in a 3D game. The amount of displacement along the length of the stroke can then be varied, and a "stroke texture" can even be applied to the object. Since we have already determined that the edge is in view at this point, very few additional polygons are added to the render using this method.

Listen Up, You Degenerate

I said earlier that this system works for closed, regular meshes. That means that each triangle edge is connected to exactly one other triangle. There are no edges that are only connected to one triangle and no edges connected to more

than two triangles. In most cases, this is a good thing to create for a variety of reasons. It is just good 3D modeling, and you should encourage all 3D artists to create models this way. However, developers live in the real world. Not all models are created equal. You will run into degenerate triangles in models.

How do you deal with this? The easy answer is to identify where the problem is and have the artist fix it. This may not always be possible, so we should deal with the problem. If an edge has only one triangle attached, what should I do? It is likely that this is something we would like to draw if it is visible. So, one approach is to mark it as "draw if seen," just as I did with the hard edges. The other approach is never to draw those edges. Whatever looks best for your model will work fine.

For edges that connect to more than two triangles, the easiest solution is to add a second entry in the edge database for the extra triangle and connect it to one of the other triangles. Or, to be more thorough, you could have three entries in the edge database to cover all possibilities. It will all work out, so don't sweat it too much. While degenerate triangles may be a bigger problem in other algorithms that use silhouettes, such as shadowing, for this application it shouldn't matter.

Adding Some More Detail

I can still combine the stroke rendering system with another system for filling in the details of the model. In fact, I really need to draw the object to some extent if I want it to look exactly correct. The silhouette algorithm detects edges even if they are behind or covered by other parts of the object. As long as they are silhouettes relative to the viewer, the system will draw them. Since everything is drawn with its correct Z information, I can use the Z-buffer to cover up the parts that should be covered.

I can simply render the triangles of the object first, writing into the Z-buffer but not writing any color information. Or, another option is to actually draw the fill colors. For a cartoon system with this type of stroke line, you will want to draw the paint colors.

For the cave painting example in Figure 3, I used my cartoon painting system to draw shading slightly translucent on only the highlights. I then set the rest of the paint to render completely transparent. This writes to the Z-buffer but doesn't change the colors, and the stroke comes out correct.

Climbing Out of the Dark

Now we have a nice system for rendering artistic-style drawings with multiple ink strokes. There are some places where we can do some more work. The current system looks at all the potential edges every frame. This could be a bit costly for a large detailed model. A better approach would be to store the results from the previous frame and test those edges first.

Another optimization method would be to precompute which edges are visible from various angles and store them in index lists. Then, by comparing the current view angle with the precomputed lists, the amount of testing needed could be greatly reduced.

Once an optimized method for computing silhouette edges is created, it can be used for things other than artistic rendering styles. The computation of an accurate silhouette is also very helpful when trying to shadow objects in a 3D environment — but that will have to be the topic for another day. Until then, explore the sketch rendering system by grabbing the demo and source code off the *Game Developer* web site at www.gdmag.com.

FOR MORE INFORMATION

 Northrup, J.D., and Lee Markosian. "Artistic Silhouettes: A Hybrid Approach." *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering* (NPAR 2000). pp. 31–37. See www.cs.brown.edu/research/graphics/research/art/artistic-sils.

 Curtis, Cassidy. "Loose and Sketchy Animation." SIGGRAPH 98 Technical Sketch. See www.cs.washington.edu/homes/cassidy/loose.

Building a Game Font

Ohhh, the pain . . . O.K., it's not that bad, but building a game font isn't usually at the top of most game artists' "favorite things to do" list. It is something that will certainly take up a portion of time in the development cycle. Depending upon the complexity of the user interface and the requirements of the game, it can balloon into one of those tasks which seems to eat up days of your production schedule. A logical approach to determining the needs of your game and a good production plan can make the process much more palatable.

So what is a game font? Well, it loosely describes a set of cels that translate to a specific set of keyboard inputs, allowing for text or keyboard-based input and display in a game environment. Some games will bypass the use of cels by utilizing a TrueType font for in-game interfaces. TrueType is a digital font format that was originally designed by Apple Computer. It is a vector-based file that is efficient in storage and processing. One of the key things it contains is "hinting" information, allowing the designer to give hints on defining which pixels are used to create the letter form at very small sizes and create the best character bitmap shape. (I'll talk more about hinting later in the column.)

One of the things that a TrueType font doesn't do is give you the ability to have a multi-colored font, or a font with a drop shadow or any number of bitmap tricks that can be created in a paint program. In order to explore this, let's take a closer look at TrueType fonts and some of the ramifications you might need to think about before choosing them for your game.

I Found It . . . It's Free!

Why wouldn't you simply choose or build a good TrueType font for your game and be done with it? Well, there are several things to consider before going down that path. One of the first issues to become familiar with is the copyright on the font. While it is easy to download hundreds and hundreds of TrueType fonts off the Internet in a matter of an hour or two, that doesn't constitute legal ownership.

Many of the fonts available on the Internet are specifically copyrighted, and the owners want a piece of the action. The last thing any developer wants is to have a suit leveled against them for using a custom font that they were sure was a freebie based upon the information they got off of "www.these_fonts_are_free.com." One of the big problems with these readily accessible fonts is that their origins may be difficult to track. Don't assume that a font is free,

even if you've found it on several font web sites under the free section. While some sites make an effort to be legal and fair, some do not. I discovered in one case, after some research, that a font I was interested in had originated from a company whose sole business was to create, copyright, and sell custom fonts. None of the sites I visited indicated the font was anything other than free.

Another problem with a large percentage of the "free" fonts on the Internet is that they don't really do you much good when it comes to localization. Many of those that are available are simply display type fonts which contain the numbers and upper- and lowercase letters, but not much else. The characters needed for localization in other languages are oftentimes left out. You may find that your free font needs to have a large number of characters added.

Localization Considerations: Those Pesky Umlauts and Other Doohickeys

One of the primary goals of a game font is to allow the product to be localized or converted to a foreign language with as little pain as possible. Sure, it's easy enough to make the text fit in the right spots in English, but what happens in the German translation? How about French? Italian?

The other big advantage of a text-driven font is flexibility. Imagine a game GUI where you've made 30 custom-sized buttons with the word "accept" embedded into the artwork. If the designer decides to change the word to "O.K.," you've got some Photoshop work to do. If, on the other hand, the text is code-driven and placed over the background artwork, then a 15-second search and replace in a text file is all that is needed to update the design.

One thing you should consider avoiding is text that is incorporated or embedded into the art. The main reason is that each asset that contains text will need to be tracked and touched for every language conversion. This may not seem to be that big of a deal at first, but if your game ends up being a big hit, you can bet that



MARK PEASLEY | Mark hangs his helmet at Gas Powered Games, where he's the art director on a real-time 3D RPG called DUNGEON SIEGE. Drop him a line at mp@pixelman.com or visit his web site at www.pixelman.com.

the marketing department will want to leverage it into every possible country. This generally means that as soon as the game is released to market, the localization process will begin in earnest. The more custom rework there is to be done, the longer each language conversion takes, and the more chance there is for errors to be introduced into your artwork. As someone who once converted an entire interface to Japanese (I don't speak or write the language), I can assure you that this is not a pleasant task.

As a rule, try to keep most of the interface and game graphics free of embedded text. In some cases, however, it is unavoidable. If you limit such occurrences as much as possible, you will be better off. If you do embed the text, keep an up-to-date spreadsheet of which assets contain text. This will make tracking down those pesky files much easier. Also, keep your base Photoshop files in layers, with the text separated in its own layer. This will make it easier for the artist involved with converting the files to the new language. As an added bonus, it allows you just a bit more control over how much alteration occurs to the art. If all someone is doing is replacing a text layer, then they won't have much of a chance to degrade or alter your textures.

Character Sets, Single-Bytes, and All That Technical Stuff

In order to explain a little about what's going on under the hood of a game font, we need to understand some of the basics. So what exactly is a character set? In technical terms, it is a set of glyphs or shapes that represent the letters which make up a font. Certain languages have specific glyphs that make up the necessary shapes for that font to be read.

Characters are represented by what are known as character codes. These codes are generated and stored when a user inputs (using the keyboard) a document. Most of the Western European translations can be covered by the use of a single-byte character set which provides 256 character codes (see Figure 1). This set generally contains the Latin letters, Arabic numerals, punctuation, and some drawing characters.

However, 256 characters falls well short of what is necessary for users of a single font for the Far East, which may need as many as 12,000 characters. In order to provide the necessary character codes, double-byte or multi-byte character sets (DBCS/MBCS) are used. These sets are a mixture of single-byte and double-byte character encoding and provide more than 65,000 character codes.

Unicode is a 16-bit encoding method which covers many of the characters used in general text interchange throughout the world. All Unicode values are double-byte, which simplifies the way a Unicode-based system reads a string of text. While it is easier to deal with than the multi-byte character sets (according to a programmer I know), it is incompatible with the APIs of Windows 95 and Windows 98. Because of this, it may be a few years before a 100 percent Unicode game will come out.

0	32	@	64	'	96		128		160	Á	192	à	224	
1	!	33	A	65	a	97		129	i	161	À	193	á	225
2	"	34	B	66	b	98	cc	130	c	162	Â	194	â	226
3	#	35	C	67	c	99	f	131	£	163	Ã	195	ã	227
4	\$	36	D	68	d	100	..	132	¤	164	Ä	196	ä	228
5	%	37	E	69	e	101	...	133	¥	165	Å	197	å	229
6	&	38	F	70	f	102	†	134	§	166	Æ	198	æ	230
7	'	39	G	71	g	103	‡	135		167	Ç	199	ç	231
8	(40	H	72	h	104	^	136	"	168	È	200	è	232
9)	41	I	73	i	105	%	137	©	169	É	201	é	233
10	*	42	J	74	j	106	„	138	ª	170	Ê	202	ê	234
11	+	43	K	75	k	107	<	139	«	171	Ë	203	ë	235
12	.	44	L	76	l	108	Œ	140	¬	172	Ì	204	ì	236
13	-	45	M	77	m	109		141		173	Í	205	í	237
14	.	46	N	78	n	110		142	®	174	Î	206	î	238
15	/	47	O	79	o	111		143	-	175	Ï	207	ï	239
16	0	48	P	80	p	112		144	°	176	Ø	208	ø	240
17	1	49	Q	81	q	113		145	±	177	Ñ	209	ñ	241
18	2	50	R	82	r	114		146		178	Ò	210	ò	242
19	3	51	S	83	s	115	"	147		179	Ó	211	ó	243
20	4	52	T	84	t	116	"	148	’	180	Ô	212	ô	244
21	5	53	U	85	u	117	•	149	µ	181	Ö	213	ö	245
22	6	54	V	86	v	118	-	150	¶	182	Ö	214	ö	246
23	7	55	W	87	w	119	-	151	‘	183		215	÷	247
24	8	56	X	88	x	120	-	152	,	184	Ø	216	ø	247
25	9	57	Y	89	y	121	™	153		185	Ù	217	ù	249
26	:	58	Z	90	z	122		154	°	186	Ú	218	ú	250
27	:	59	[91	{	123	>	155	»	187	Û	219	û	251
28	<	60	\	92		124	œ	156		188	Ü	220	ü	252
29	=	61]	93	}	125		157		189		221		253
30	>	62	^	94	~	126		158		190		222		254
31	?	63	_	95		127	Ý	159	¿	191	ß	223	ÿ	255

FIGURE 1. An example of an ANSI character code chart that contains the characters used for Western European translation.

Building Your Own TrueType Font

After determining that you will be using a single-byte character set, you will most likely consider creating your own TrueType font. After all, that would solve your copyright issue and give you the creative freedom to customize your font as needed, right? (Insert dramatic pause here.) Well, you will want to consider this carefully.

Creating a useful TrueType font is no small undertaking. Font design is an art in itself and requires a lot of attention to detail as well as a firm understanding of good design principles. Rather than mathematical precision, a well-designed font relies on visual equality. Each letterform is considered for its balance, and subtle alterations are made to create a visual equality among the letters. In addition, for the font to be usable and, more importantly, readable at the lower resolutions common in game interfaces, it will need to have hinting.

All of these subtleties take time and a keen eye in graphic design. In addition, you will need to get a font-specific program such as Macromedia Fontographer in order to create the font easily. Once you have it, you will need to spend some serious time not only familiarizing yourself with the various specialized glyphs in your

character set, but also getting a firm control of the program. Once you've gotten to the stage of actually creating the letterforms, you will then need to start considering kerned pairs and hinting.

Hinting

So what exactly is hinting and what does it do? The easiest way to think about it is to consider what happens to a font when it is rasterized or converted to a bitmap for display on your screen. The vector information is converted to a bitmap and anti-aliased to provide a smooth line that defines the letter. This is a fairly straightforward process until the font size is reduced down to a small size. Once you get into the 10- to 14-point range, the definition of where the lines go is more difficult for the program to determine when it converts the vector information to a bitmap. At a certain point, a choice needs to be made concerning where the pixel is placed on the underlying rasterization grid. This is where hinting comes into play.

A hint is technically a mathematical instruction added to the font that distorts the character's outline before it is converted to a bitmap. These modification hints allow the designer to have a fairly high level of control over the resultant bitmap shape of the letter.

Without this type of control, features that define the font (line weight, widths, serif details) can become inconsistent, irregular, and sometimes even missed entirely. This can have a dramatic effect on how legible the font is. In Figure 2, you can see how the letter would reduce down if it were simply scaled to the appropriate size and rasterized. The second image shows how the hinting alters the letter shape so that the placement of the pixels is controlled. This, in turn, causes the font to be much more legible at a small size.

As you might guess, this sort of customization on a letter-by-letter basis can become a substantial time sink. If you are dealing with a larger publisher, they may already have purchased rights to use some fonts in their products. This doesn't lend itself to a custom look, but it can save time and money if the font isn't too critical of an element in the overall design.

Building a Bitmap Font

Another alternative is to consider building a bitmap font. These are usually created by rasterizing a font, which can contain multiple colors and alpha transparency. The font is then divided into cels which are linked to the character code via programming code. The good news is that it's done in a paint program and is a fairly straightforward process.

There are a couple of different ways to create your font. The first is simply to make a character set that contains every character needed for translation to the Western European countries. This entails rasterizing each of the 256 characters and laying them out in a base file. The exact layout is something you will be working directly on with the UI programmer. A pixel is usually used to define one side or corner of the cel that encloses the letter (see Figure 3). These cels are then linked to the character codes.

If your game is a graphics-hardware-only solution, then consideration must be given to laying out the various characters so that

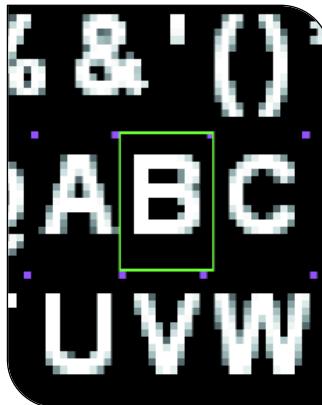
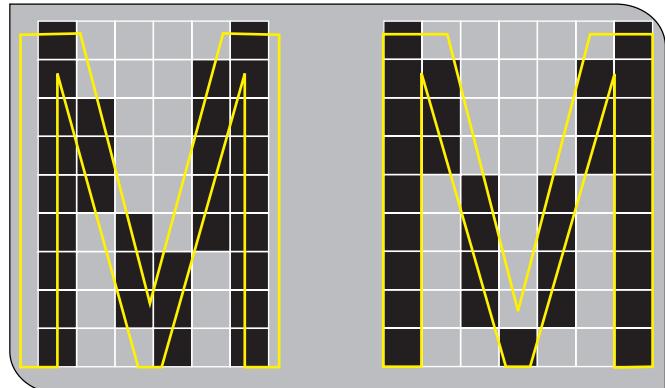


FIGURE 2 (above). This is an example of how hinting works. The first character image on the left hasn't been fitted to the underlying grid, giving a poor pattern and an awkward letter. However, the second image on the right is fitted to the grid, resulting in a balanced letterform.

FIGURE 3 (left). A close-up of the cel that defines the "B" character.

they fit on power-of-two texture maps. Again, this may be handled via a cut-up routine that the programmer establishes, or you may need to lay them out on a custom basis.

In our example font (see Figure 4), each of the 256 character cels needs to be accounted for and laid out on a 256×256 texture map. The first 32 slots are blank, so oftentimes you can simply start with cel 32 and continue from there.

Another method that can save on space is to have a text file that is linked to the bitmap file. The text file contains nothing more than the string of characters in the same order as they are found on the bitmap font file. The advantage to doing it this way is that the precise order of the cels need not be adhered to. It also allows you to add characters as needed without breaking the code. This is also an excellent way to deal with multiple font sizes if you only need a few specific letters. The downside is that organization begins to go out the window, and tracking two assets that need to maintain an exact match can quickly turn into more of a headache than you might think.

Leading and Kerning in a Bitmap Font

One of the disadvantages of a bitmap font is that kerning (the spacing between letter pairs) is usually not done. Because kerning requires quite a bit more coding to accomplish, it usually ends up on the cutting room floor. When dealing with a cel, you can define how much spacing a specific letter gets on one side, but



FIGURE 4. An example font laid out to fit onto a 256×256 texture map. Note the pixels that define the cel borders. These are usually in a unique color so that the code can recognize the border pixels.

you won't be able to get a customized kerning look. What this means is that your font will be generically spaced, usually with one or two pixels between each letter, regardless of what's next to it.

While this isn't the ideal solution visually, it is one of those things that we've become accustomed to seeing in games. If you have some free time on your hands, and the UI programmer wants to get fancy, you can make specific kerned pairs of letters, such as lowercase double *ls* (ll) or double *os* (oo). The code is then set up to detect any of the unique letter combinations that you have custom spaced, and replaces the standard-spaced letters with your kerned pair.

Hopefully, this has given you a bit of insight into creating a game font. I've done quite a few of them over the years, and each time it's a bit different. Each program and each programmer working on the UI approaches the problem differently from the production standpoint. If you take the opportunity to plan the process out and do some research first, it can save you a lot of wasted time later.

FOR MORE INFORMATION

Macromedia Fontographer
www.macromedia.com/software/fontographer

Microsoft Typography
www.microsoft.com/typography

TrueType Font Specification
www.msdn.microsoft.com/library/toc.asp?PP=/library/toc/specs/specs12.xml&tocPath=specs12

Fast AGP Writes for Dynamic Vertex Data

M

any games being written today take advantage of graphics cards that support hardware transformation and lighting when available. But some of these games also have vertex data that is modified dynamically for performing techniques such as multiple-bone skinning of meshes. Ironically, when modifying the vertex data dynamically on a per-frame basis, a significant speedup can sometimes be gained by modifying more of the data. In this article we'll take a look at accelerated graphics port (AGP) memory and the interaction between the processor, the chipset, and the graphics card. In the process, we'll learn about write-combining memory and how to avoid partial writes which can slow down your AGP updates. When we're finished, you'll have an understanding of these terms and how to use this information to achieve performance increases of up to 20 percent or more in your games. Let's start by examining what AGP memory is.

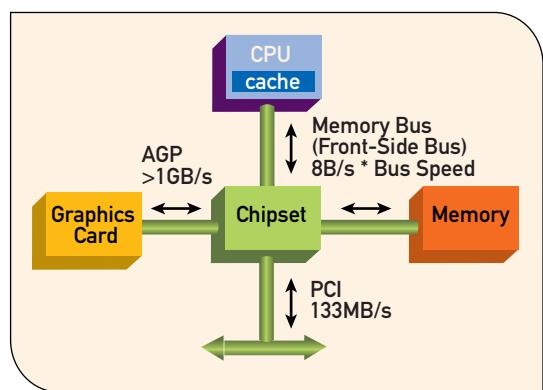


FIGURE 1. PC architecture with AGP 4x support.

DEAN MACRI | Dean is a technical marketing engineer within the Solutions Enabling Group at Intel. He enjoys working with mathematical aspects of 3D graphics and performance optimization techniques. Dean welcomes e-mail on anything 3D-related and can be reached at dean.p.macri@intel.com.



Illustration by Juan Alvarez

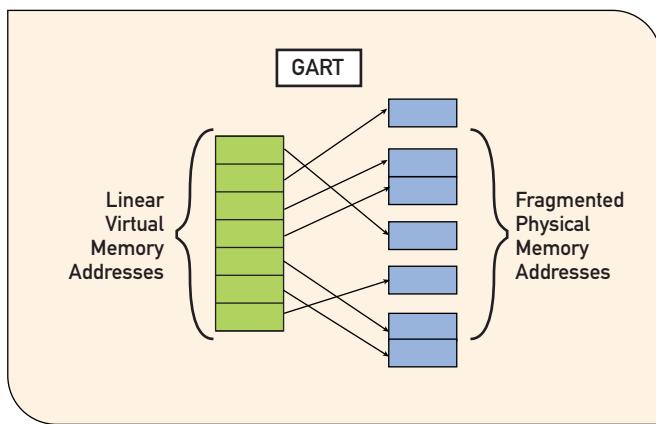
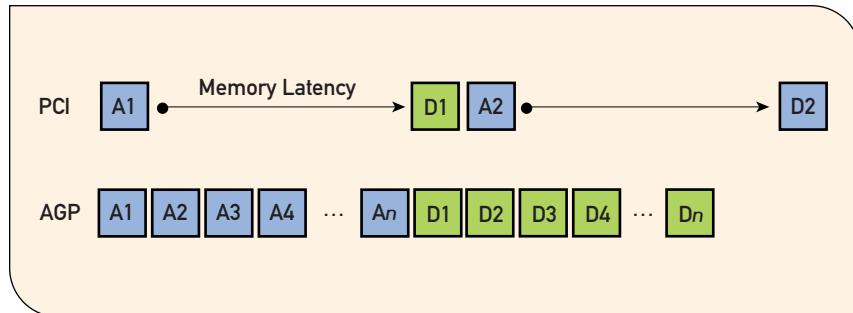


FIGURE 2 (top). AGP pipelining versus PCI transfers.

FIGURE 3 (bottom). Remapping of memory addresses through the GART.

AGP History

Introduced with the Intel Pentium II processor in 1997, the AGP interface provides a high-speed mechanism for video cards to access main system memory. Debuting at 266MB/s peak transfer speed, the current specification, known as AGP 4x, allows for a peak transfer rate of 1,067MB/s, eight times faster than the standard PCI bus found in desktop systems. Figure 1 depicts the architecture of a PC with support for AGP 4x. Notice that the graphics card has a direct pathway along the AGP bus, through the chipset to the system memory. For a system with a 133MHz front-side bus, the AGP bus speed is equivalent to the memory bus speed but still falls short of the 3.2GB/s peak bus speed of Pentium 4 processor-based RDRAM systems.

In addition to the high transfer speed, requests on the AGP bus can be pipelined, whereas the PCI bus only supports sequential transfers with a special case for bursting. Figure 2 shows the benefit of pipelined access as provided by the AGP specification. The PCI bus requires that the data D1 arrive from the memory request for address A1 before it can submit a memory request for address A2. In a more efficient manner, the AGP bus can accept memory requests for addresses A2 through An while waiting for data item D1 to arrive. In this fashion, data items D2 through Dn arrive much sooner than in the nonpipelined scheme such as that provided by the PCI bus. Something we'll talk about later is a burst operation that both the AGP and the PCI buses support. In a burst operation, a small number of data items (four to eight) can

be sent across the bus with a single memory address request.

The AGP bus also provides an additional eight "sideband" address lines to allow more overlapping of previous data and addresses on the main 32 wires with new addresses and requests, for further performance improvement. We won't need to know any more about the underlying protocol and addressing scheme of the AGP specification, but the description just provided should help us better understand what's going on under the hood of the machine. What will be essential is to realize that the graphics card uses the AGP interface to access data in the system's main memory. It's this data in main memory that we're going to look at in detail.

Prerequisites for AGP Memory

For applications to be able to take advantage of the benefits of the AGP bus, the operating system and the hardware must perform several housekeeping tasks. First, because system memory is being used primarily for typical applications, there can be considerable fragmentation based on what applications are running and what particular memory allocations have occurred. A request to allocate 256KB of memory for the graphics card to access via the AGP bus could result in several small, noncontiguous chunks of main memory being allocated. To make this appear as one contiguous 256KB piece of memory to the graphics card, the chipset, which controls the communication between the processor and main memory as well as the PCI bus, has something called a Graphics Address Remapping Table (GART). The GART maps a linear range of virtual memory addresses to multiple 4KB physical addresses in main memory. Figure 3 shows an example of a region of linear memory visible to the graphics card, with the memory being mapped to a non-contiguous set of pages in physical memory. The amount of memory available for remapping by the GART is often determined by a setting in the BIOS known as the AGP aperture. Values vary from system to system, but there is usually a maximum limit of half the available system memory.

We've addressed the tasks the chipset must perform. Now, because the graphics card will be accessing the memory regularly and performance is critical, the operating system must make sure that memory which will be accessed by the AGP bus is not swapped out to disk. The operating system does this by locking the pages of memory. Finally, because current processors have one or more levels of cache to improve performance of typical applications, something must be done to make sure the graphics card sees the most up-to-date data. To achieve this without requiring the graphics card to "snoop" the caches of the processor, the memory being used for AGP transfers is marked as not cacheable, or uncachable, by the processor. This uncachable memory is the heart of what we're going to look at to achieve performance improvements in 3D applications. To understand it, we need to examine a feature which Intel processors have carried for several generations.

Memory Type	Cacheable?	Fill Cache on Read?	Fill Cache on Write?	Use Write-Combining Buffers?
Uncacheable (UC)	No	N/A	N/A	No
Write-Combining (WC)	No	N/A	N/A	Yes
Write-Back (WB)	Yes	Yes	Yes	No
Write-Through (WT)	Yes	Yes	No	No

Table 1. Common memory types selectable with MTRRs.

Memory Types

Starting with the Pentium Pro processor, Intel processors have provided a mechanism for changing the access characteristics of regions of memory via a set of registers known as the Memory Type Range Registers (MTRRs). These registers are not accessible to a normal application and can only be changed by the operating system or a device driver. However, the drivers that support AGP memory use the MTRRs to provide the right characteristics to the memory being accessed by the graphics card. Let's look at the various options that can be selected using the MTRRs.

Table 1 shows the most common memory types used by current systems. Most of the memory in a system is marked as cacheable memory, with one of two schemes for synchronizing cache memory with main memory: write-back or write-through. With write-back cacheable memory, both read and write cache misses cause cache line fills. However, write-back memory only writes cache lines back to main memory when they are being evicted due to contention with another memory access or explicitly by being invalidated by the processor. Write-back memory provides the best performance possible, because in many cases the processor can do many operations on the data in the cache before ever writing out to main memory. However, this requires that any device with access to main memory be able to snoop memory accesses in order to maintain system memory and cache coherency. Write-through memory still performs cache line fills on read misses but never on write misses. All writes are written to a cache line (assuming a cache hit) and through to main memory.

Interesting things happen when memory is marked as uncacheable. All reads and writes go directly to system memory, bypassing the caches. As you can imagine, this slows down both read and write accesses, leading one to wonder why memory would ever be marked as uncacheable. Let's answer this question by revisiting something we briefly touched on previously. Sometimes, devices other than the processor need to fetch data from main memory. The obvious example would be the graphics card fetching data from main memory via the AGP bus. In doing so, there needs to be a way to make certain that data writes done by the processor appear in the actual main memory, not just in the data caches to

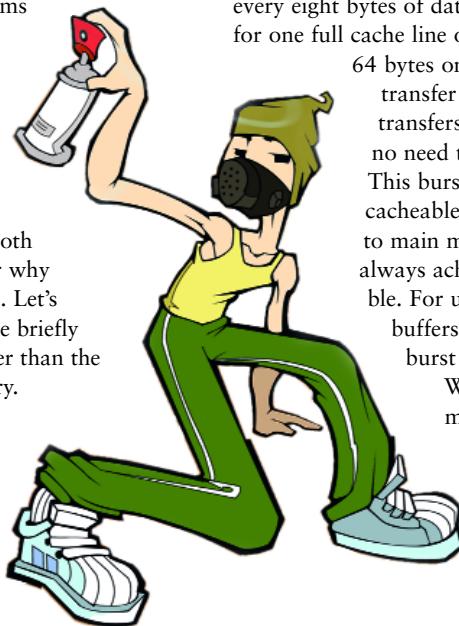
which only that processor has access. Additionally, the graphics card may also write to main memory, so the processor needs to be certain that if it reads from the memory, it sees the actual data written by the graphics card and not some old data stored in the data caches. By marking memory as uncacheable, we can be certain that the data the graphics card sees is identical to what the processor sees and vice versa.

To further complicate things (but to improve performance in some cases), memory can be marked using a fourth type: write-combining. Write-combining memory is uncacheable but with a twist. When writing to write-combining memory, there are a few buffers internal to the processor where data writes are temporarily stored. The number of buffers varies with each new processor, with four on the P6 family of processors (Pentium Pro, Pentium II, and Pentium III) and six on some later versions of the Pentium III processor and on the Pentium 4 as well. More buffers means performance improves, because you can disperse your writes a bit without causing an eviction. Let's take a closer look at these write-combining buffers to see what advantages they bring to the table.

Write-Combining Buffers

When discussing the AGP bus, we mentioned burst operations supported by the PCI bus and the memory bus. In a burst operation, rather than sending one memory address for every eight bytes of data, it's possible to send one memory address for one full cache line of data (32 bytes on the P6 architecture and 64 bytes on the Pentium 4 architecture). The data transfer still only happens on a 64-bit bus, but the transfers can happen more quickly because there's no need to wait for additional memory addresses. This bursting operation is used by write-back cacheable memory when it flushes a cache line out to main memory, so write-back cacheable memory always achieves the highest write throughput possible. For uncacheable memory, the write-combining buffers offer a means to achieve benefit from burst transfers of data.

When software writes to memory that is marked as write-combining, the processor will begin to fill one of the write-combining buffers (remember there are four or six of these). It's possible to think of the write-combining buffers as a very small, fully associative cache that is never filled



LISTING 1. D3DVERTEX structure from DirectX 7.

```
typedef float D3DVALUE;
typedef struct _D3DVERTEX {
    D3DVALUE x; /* Homogeneous coordinates */
    D3DVALUE y;
    D3DVALUE z;
    D3DVALUE nx; /* Normal */
    D3DVALUE ny;
    D3DVALUE nz;
    D3DVALUE tu; /* Texture coordinates */
    D3DVALUE tv;
} D3DVERTEX;
```

LISTING 2. Creating the sample vertex buffer in AGP memory.

```
pDx8Device->CreateVertexBuffer(
    mVertexSize*mNumVertices,
    D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY,
    D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1,
    D3DPPOOL_DEFAULT,
    &mpVertexBuffer);
```

from main memory. As you write data, the buffers are filled, and you can even read back the bytes just written without doing a fetch from main memory. You can also rewrite bytes that you've already written without causing an eviction. When you try to write data to a memory address that isn't represented by one of the buffers, however, one of the existing buffers will be evicted to main memory. This is where the magic happens. If every byte in the buffer (32 bytes for P6 processors, 64 bytes for the Pentium 4 processor) has been modified, then the buffer is transferred to main memory via one burst transaction. If not every byte has been modified, then the processor writes the buffer out eight bytes at a time.

You're probably beginning to see that if you modify 31 consecutive bytes and don't modify the 32nd, then it will take four bus transactions on P6-family processors to evict the data. The same eviction could have happened with one transaction if you had just written data to the 32nd byte (of course, you'll want to write the appropriate data for your application). On Pentium 4 processors, the worst case is even worse in terms of the number of transactions. If you modify 63 bytes rather than 64, it will take eight bus transactions rather than one — ouch. Fortunately, the high speed of the bus on Pentium 4 processor-based systems alleviates some of the pain. What we see, though, is that to achieve maximum performance, it's essential you modify every byte of a write-combining buffer.

The last thing to mention about write-combining memory is that it is weakly ordered versus strongly ordered uncachable memory that isn't write-combining. Weakly ordered means that if you write out bytes 1, 4, 3, and 2 in that order, they won't necessarily get written to main memory in that order. In fact, they probably won't. This is due to the buffering performed by the write-combining buffers. Weakly ordered memory is fine for the

types of data being sent across the AGP bus. For devices that perform memory-mapped I/O, however, writes would need to be strongly ordered so that they appear on the bus in the exact order that the program wrote them. To guarantee that all the write-combining buffers get written to main memory, several instructions are available, including SFENCE and MFENCE, CPUID, and a few others. Rest assured that the API (for example, Microsoft DirectX) that gives you the AGP memory will manage the task of making sure that all of your write-combining transactions have been committed to memory before allowing the AGP device to read the memory.

Experimenting with Write-Combining Memory

We've covered enough of the low-level details of AGP and write-combining memory. Now let's see how to create a sample application which demonstrates the benefits of ensuring that the write-combining buffers are utilized properly. We'll also see the deleterious effect of causing multiple bus transactions due to partially filled write-combining buffers. The techniques I'm going to describe here are specific to DirectX 8 under Windows 98. There seems to be something unexpected happening behind the scenes on Windows 2000 which may be related to the operating system, drivers, or both, so I won't cover that here. I'm also assuming that you understand how to use the DirectX APIs. If not, you can find the essential information at www.microsoft.com/directx.

In a DirectX 8 Direct3D application, three basic types of data resources can be allocated in AGP memory: vertex buffers, index buffers, and textures. Under DirectX 7 it was possible to request AGP memory specifically. Using DirectX 8, however, there are some guidelines that the application can make, but the API and drivers ultimately determine where the resources are allocated. For our example, we're going to be allocating a vertex buffer that we would like to reside in AGP memory. To do this, we must take care of several things. First, we need to be creating a vertex buffer that will be used by a graphics card which supports hardware transformation and lighting (currently, only the GeForce cards from Nvidia and the ATI Radeon card fulfill this requirement). Otherwise, the DirectX API won't allocate the vertex buffer in AGP memory, because it would severely limit the performance of the transformation and lighting pipeline. So, when we use the CreateDevice() method of the ID3DX8 interface, we need to specify either D3DCREATE_MIXED_VERTEXPROCESSING or D3DCREATE_HARDWARE_VERTEXPROCESSING to be able to have the graphics card do the transformation and lighting of the vertices.

Next, we need to include the usage flags D3DUSAGE_DYNAMIC and D3DUSAGE_WRITEONLY when calling the CreateVertexBuffer() method of the IDirect3DDevice8 interface. These flags indicate, respectively, that we're going to be changing the data in the vertex buffer regularly and that we'll only be writing to the buffer. Additionally, we need to tell DirectX to allocate the buffer in the default memory pool (D3DPPOOL_DEFAULT). With these restrictions, we should get a vertex buffer allocated in AGP memory. For our example, we'll use a vertex structure that contains information for vertex position, normal,

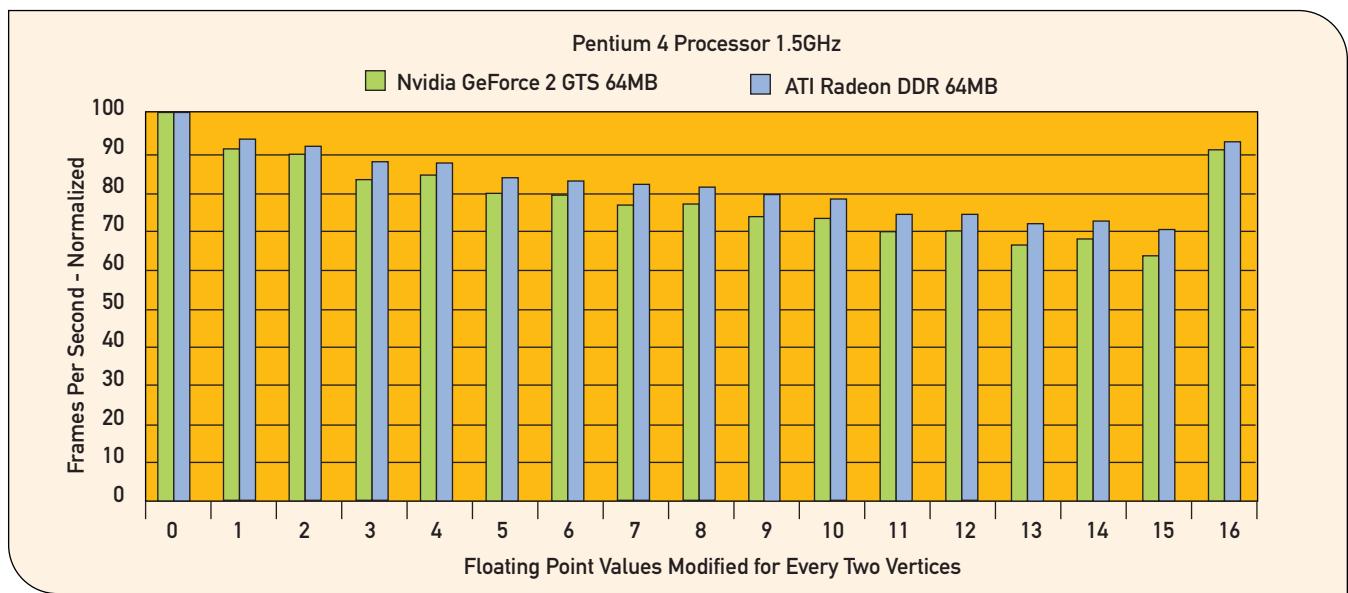
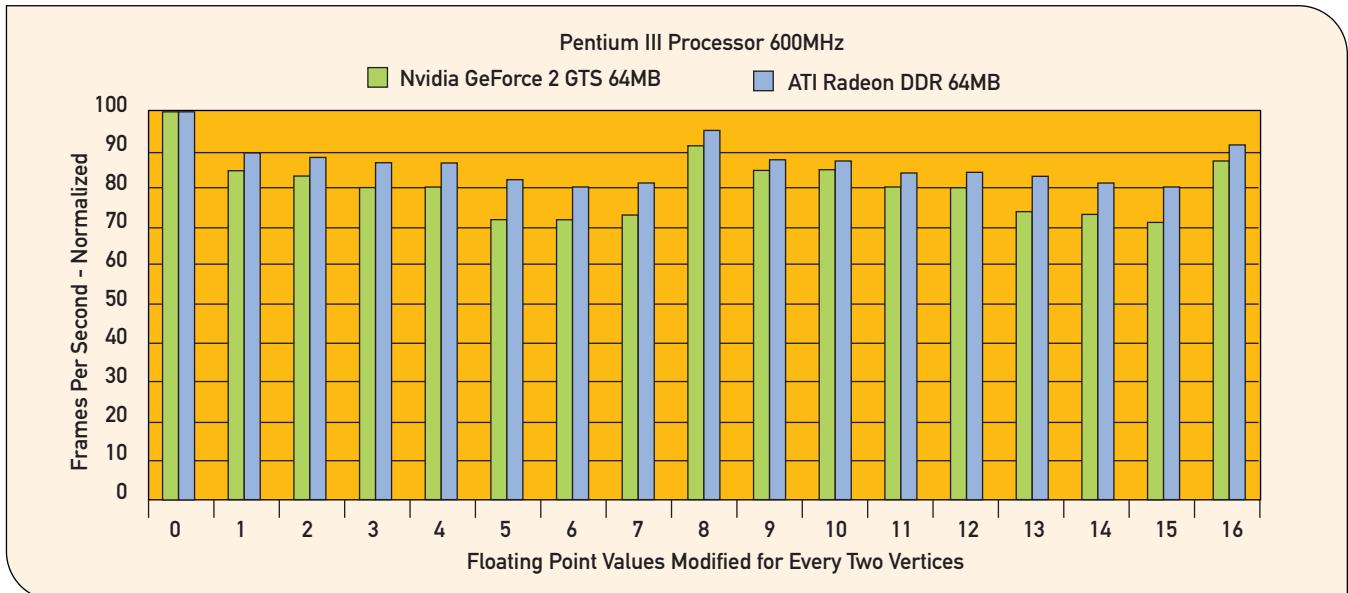


FIGURE 4 (top). Test application results on a Pentium III processor with two different graphics cards.

FIGURE 5 (bottom). Test application results on a Pentium 4 processor with the same two graphics cards.

and one set of texture coordinates, as shown in Listing 1. I've chosen this format for two reasons: it's the `D3DVERTEX` structure that was used by previous versions of DirectX, and more importantly, it's exactly 32 bytes, which is the size of one cache line on P6-family processors and half a cache line on the Pentium 4 processor.

Listing 2 shows a code snippet for creating a vertex buffer using the flags just described. In the example, `pDx8Device` is a pointer to the `IDirectX8Device` interface, `mVertexSize` is the size of the vertex type (32 bytes), `mNumVertices` is the number of vertices the buffer will be able to hold, and `mpVertexBuffer` will contain the pointer to the `IDirectX8VertexBuffer` interface when the function returns.

When we lock the vertex buffer, the pointer we receive should

point to memory that is uncachable and write-combining. Using the memory pointer returned from the `Lock()` method of the vertex buffer, we can now experiment with modifying the data in the vertex buffer. I created a simple program that renders a single, highly tessellated sphere using a vertex buffer allocated in AGP memory. At every frame, I lock the vertex buffer that holds the vertices of the sphere and move the positions of the vertices along the direction of their normal using a simple sine wave. Granted, this same task could have been accomplished using a scale, but I just wanted a simple test to experiment with the performance impact of vertex buffers in AGP memory. This program is available on www.gdmag.com. Using a copy of the original

vertex data for the sphere stored in system memory, I create a temporary set of modified coordinates and then update the positions of the data in the vertex buffer. To do this, I lock the vertex buffer at every frame. Then, I calculate new X, Y, and Z positions for the vertices one at a time and store these in a temporary vertex structure along with the original normal vector and texture coordinates for the vertex. Finally, depending on a variable that indicates the number of floats (4-byte values) to touch, I use a small piece of assembly code that uses the repeatable string move operation on **DWORDs** (**rep movsd**) to copy the system memory copy to the actual vertex buffer. This enables me to experiment with touching one float (just the X coordinate of the vertices), two floats (X and Y coordinates), and up to eight floats (the entire vertex structure). I also provided a mechanism to skip every other vertex. Because the size of the write-combining buffers on the Pentium 4 processor is 64 bytes, this enabled me to see what happens when half of the 64-byte buffers are modified but not the entire buffer.

Results

Using the test application just described, I used the displayed frame-rate counter to do a visual comparison with different parameters. Figures 4 and 5 show the results on two different systems with two different graphics cards. The X-axis of the charts shows how many consecutive floating-point values are modified per vertex. The Y-axis of the charts shows the frames per second, normalized according to the frame rate when none of the vertex data is modified. As you can see from the charts, the frame rate drops dramatically with the number of bus transactions required to write out a partially filled write-combining buffer. But when the buffers are completely filled, performance picks up. You can also see the size of the write-combining buffers, because on the Pentium III system we see performance pick up when eight floats (32 bytes) and 16 floats (64 bytes) are modified. On the Pentium 4 system, we only see performance pick-up when 16 floats are modified.

Checking for Partial Writes Using Vtune Analyzer

I also wanted to define a more rigorous test so that you, as a game developer, could have somewhere to go to see if your application is suffering from partial writes of write-combining buffers. To do this, I used the Intel Vtune Performance Analyzer 5.0. There isn't space here for a full tutorial on using Vtune Analyzer, but I'll give a quick description of how to use it to check for partial writes of write-combining buffers. For more information on Vtune Analyzer, see the For More Information box.

When running Vtune Analyzer on a P6-family processor or Pentium 4 processor, it's possible to count events that occur in the processor. (Note: mobile processors — those found in laptops — don't support the event-counting mechanism.) The most common



event to count is the number of CPU cycles or clock ticks that have elapsed. Other events include the number of instructions executed, number of L1 cache hits, and many more. For P6-family processors, we're interested in the "External Bus Partial Write Transactions" event. This event occurs, obviously enough, every time a partial write happens on the bus. Because all writes to write-back, cacheable memory are performed one complete cache line at a time, this event typically only occurs when a write to uncacheable memory

has happened. On the Pentium 4 processor, an event specific to write-combining memory, "Write WC Partial," is the one to check. Using these events, you'll be able to determine if your application is causing a significant number of partial writes to occur.

Wrapping Up

We've seen the performance implications of partial writes to write-combining memory and learned how to avoid the problems just by moving a little extra data, which is pretty counterintuitive. I encourage you to examine programs where you modify vertex buffer data on a per-frame basis and see if you may be causing partial writes to write-combining memory. The speedup can be significant, as we've seen. And the same techniques apply to other resources allocated in AGP memory. If you're modifying textures regularly and seem to be touching data a few bytes at a time, then maximizing your use of the write-combining buffers could improve your performance considerably. ☺

ACKNOWLEDGEMENTS

I'd like to thank Pete Baker, Kim Pallister, and Will Damon for their valuable input into this article.

FOR MORE INFORMATION

Microsoft DirectX
www.microsoft.com/directx

Intel Developer Site
<http://developer.intel.com>

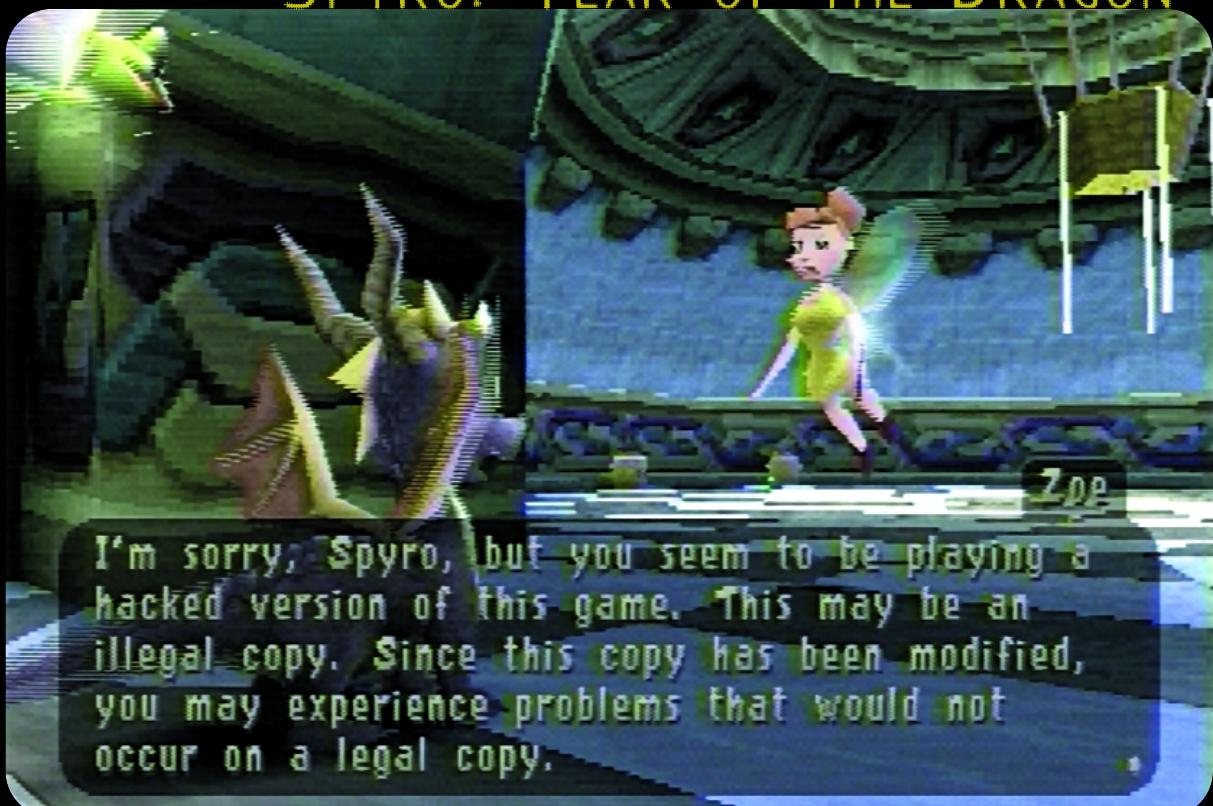
Intel Vtune Performance Analyzer
<http://developer.intel.com/software/products/vtune>

Nvidia GeForce 2 GTS
www.nvidia.com/products/geforce2gts.nsf

ATI Radeon 64MB DDR
www.ati.com/na/pages/products/pc/radeon64_ddr/index.html

KEEPING THE PIRATES AT BAY

IMPLEMENTING CRACK PROTECTION FOR SPYRO: YEAR OF THE DRAGON





o you've worked 10- to 12-hour days for the past two years, trying to make your latest game the best ever. You even added copy protection to try to stop the pirates, but within a few days of release there are already crack patches flying around the Internet. Now anyone can help themselves to your hard work, without so much as a "please" or "thank you."

This is what happened to Insomniac's 1999 Playstation release, SPYRO 2: RİPTO's RAGE. Even though it had good copy protection, it was cracked in a little over a week. So when we moved on to SPYRO: YEAR OF THE DRAGON (YOTD), we decided that something more had to be done to try to reduce piracy. The effort was largely successful. Though a cracked version of YOTD has become available, it took over two months for the working patch to appear, after numerous false starts on the part of the pirates (the patch for the European version took another month on top of that). The release of patches that didn't work caused a great deal of confusion among casual pirates and plenty of wasted time and disks among the commercial ones.

Two months may not seem like a long time, but between 30 and 50 percent of most games' total sales occur in that time. Approximately 50 percent of the total sales of SPYRO 2, up to December 2000, were in the first two months. Even games released in the middle of the year rather than the holiday season, such as Eidetic's SYPHON FILTER, make 30 percent of their total sales in the first two months. If YOTD follows the same trend, as it almost certainly will, those two to three months when pirated versions were unavailable must have reduced the overall level and impact of piracy. On top of this, since YOTD was released in Europe one month after the U.S., those two months protected early European sales from pirated copies of the U.S. version.

So why did it take so long to crack YOTD when a patch was available for SPYRO 2 so quickly? The difference was that SPYRO 2 only had copy protection, while YOTD added crack protection. The crack protection complemented the copy protection by checking for alterations to the game, rather than just making sure the game was run from an original disk. This extra layer of protection slowed down the crackers significantly, because removing the copy protection had to be done without triggering the crack protection. Basically, YOTD is booby-trapped — one wrong bit and it will blow up in your face. This article will explain the techniques that we used in YOTD, what we learned from using them, and some ideas about how to take our techniques even farther. However, I will not go into explicit detail, as most of the coding involved is relatively simple. Crack protection is more about out-thinking the crackers than out-coding them. A great advantage of any method of protection is novelty. Even a new implementation will give an advantage over simply reusing code, regardless of whether it was successful in previously delaying a crack.

GAVIN DODD | Gavin started work at Psygnosis in 1992. There he worked on many games which he probably wouldn't want to mention (except for JOHNNY MNEMONIC on Sega CD which he is perversely proud of) and most of which never saw the light of day. The only notable products he worked on at Psygnosis were COLONY WARS and COLONY WARS: VENGEANCE. In 1999 he moved on to Insomniac Games to work on SPYRO 2 and SPYRO: YEAR OF THE DRAGON. Now he is working on a PS2 game he isn't allowed to talk about, so don't even ask. Gavin can be reached at gd@insomniacgames.com.

Defining the Problem

From the very beginning we recognized that nothing is uncrackable. Many different software and hardware techniques have been used in an attempt to stop piracy; as far as I know every one of them has been bypassed or cracked. Our goal was to try to slow the pirates down for as long as possible.

First we looked at the copy protection: was there any way to reduce its vulnerability to cracking? We could call the copy protection multiple times throughout the game, making it harder to bypass. Unfortunately, the copy protection requires exclusive access to the CD for about 10 seconds, which is an eternity when you are waiting for a level to load. So that was out of the question.

Then we looked at how a typical crack is made. Most cracks for Playstation games replace the boot executable with an “intro”

“Basically, SyPRO: YEAR OF THE DRAGON is booby-trapped — one wrong bit and it will blow up in your face.”

that proclaims the prowess of the crackers and allows cheats to be activated. This intro is concatenated with a copy of the patched executable and compressed so that the total file size is no larger than the original file. The new file bears little resemblance to the original boot program. This difference gives us the opportunity to reload the boot executable sometime after startup, causing severe corruption if it has been altered in this way. As with the previous option, this solution suffers from the problem of adding to the load time. It is also vulnerable to the pirates finding some other space on the disk to hide their crack, leaving the original boot file untouched. While this solution might have slowed the pirates down for a few more days, it didn’t seem like it was the answer we were looking for.

We decided that we needed a thorough way of detecting at run time that the game had been cracked. When we could reliably determine whether the game had been modified, we could stop the game anytime we found a discrepancy. We also needed a method that didn’t require access to the disk. It should just check the code in memory, unlike the standard copy protection or our option of reloading the boot program. This would allow us to place the check anywhere in the game, making it much harder to remove.

So now we had a definite goal, an approach that should significantly improve the protection for YOTD.

Checksumming

Finding out if a block of data has changed in any way is actually pretty easy. Techniques have been used for error detection for years and are well documented. Just search for “checksum” on the Internet. For YOTD, we decided to use a CRC checksum: it’s robust, simple, and fast.

The checksum was calculated bitwise rather than using tables, as tables would be an easy point for a cracker to attack. We took care to hide and protect the checksum values as well. If these could be found and altered easily, a cracker would simply replace the checksum with a new value that matched the cracked data,

which is far easier than removing the code. For the same reason we didn’t use functions to calculate checksums, we inlined the code as much as possible. If the code was in a function, it would only have to be removed once. The inline code would have to be removed as often as it was used.

We used a few slightly different implementations to stop simple pattern searches from being used to find the checksum code. To what degree this survived compiler optimization we don’t know. To make our lives easier we made macros. These could be sprinkled around the code and mixed in with other tasks, which would make it much more difficult to spot where the checksum was being calculated.

Unfortunately, because checksums are designed to detect errors and not modification, they cannot offer full protection against modification. The checksum value for any block of data can be

made to add up to any value by modifying the same number of bits that are used for the checksum. In other words, if the checksum is 16 bits, altering 16 bits in the data can make the checksum match any value.

To deal with this, multiple checksums were applied to the same data. Each checksum used a different start offset into the data, and stepped through the data by different amounts. This meant that overlapping and interleaved sections of data were checksummed at different points, making it almost impossible to alter anything and still have all the checksums add up. I could have used different checksum algorithms for the same effect, but in this case I didn’t have time to implement more than one method.

We used the fact that altering a small number of bits can give you any checksum value to our advantage. By inserting the correct value into the middle of the data, the checksum could be made to equal any predetermined value. This meant the checksum value could be hard-coded and therefore become part of the data being checksummed. This is bewildering to even think about, let alone try to crack.

Since the game used multiple code overlays (or DLLs), they cross-checked each other as much as possible. This further reduced the chance that any section could be altered without being spotted. If any overlay noticed a discrepancy, it altered data in the core such that no subsequent checksum would be valid. This meant that if an alteration was detected in one overlay, then other overlays loaded later would know about it. This made it difficult for the cracker to spot what actually triggered the protection, as I’ll explain later.

Obfuscation

Now that the meat of YOTD’s crack protection had been implemented, it was time to move on to the second stage — slowing down the crackers as much as possible. We had already tried to make the protection as difficult as possible to understand, mixing in the checksum code with regular game code, and using

different implementations so that it would be hard to understand. We thought that if the crackers couldn't understand what we had done, it would be a lot harder for them to crack the game. We wanted to make it hard enough to reduce the pool of people capable of cracking it. If there are only a couple of pirates with enough skill to crack a game's protection, it might take them a week or two to get around to it. Unfortunately, with YOTD being such a high-profile game, this was probably wishful thinking.

Thus we wanted to make the job of cracking YOTD time-consuming and tedious. If we could just keep the crackers busy at finding the protection, that's time taken away from them working out how to remove it. Again, we were trying to reduce the pool of people available who could crack the game. Not every cracker would have enough time available to make the crack; it probably isn't anyone's day job. On this note, it's worth pointing out that for most crackers this is a hobby. If they get bored, they may well give up. We tried to make the crackers have to wade through plenty of chaff before finding the protection. There were a couple of techniques we tried to achieve this.

First of all, if the crackers know what they are looking for, they often don't even have to boot up the game to find the protection. They can simply search the disk and sometimes even edit the protection right there and then. Simply doing an XOR of as much of YOTD's code as possible before burning it to the CD means that this technique will not work. It also makes it difficult to match up data in memory with its source on the disk. We worked under the assumption that code can't be modified if it can't be found.

Second, we made it as difficult as possible to debug. If you've ever had to debug code that behaves differently when you trace through it, you know how much of a pain debugging can be. We used trace traps to make the code behave differently if breakpoints had been placed. The checksum helped with this, as any software breakpoints used would alter the checksum. Rebooting the debug-

tion is detected, the cracker would soon find and remove all the protection. However, if we wait too long to react, too much of the game would be playable even if an incomplete crack was used. To balance this, we used as many layers of protection as possible, which occurred at different points during the game. In YOTD we had four layers, including the copy protection.

The copy protection stopped the game very early. When this was removed, the game appeared to work for some time. We assumed that the crackers generally don't play the games they crack very much, they just play until the point where the protection they know about kicks in. Then they release a crack, believing it to be complete.

To play on this, we designed the game to break in ways that weren't immediately obvious. Most of the protection is "off-camera," affecting levels other than the one currently being played. In YOTD the object of the game is to collect eggs and gems, which are then used to open later parts of the game. The protection removed eggs and gems, so that the player could not make progress. We tried to make the game unplayable for any length of time, while at the same time making it difficult to determine exactly where things had gone wrong. If errors accumulated slowly until the game broke, the cracker would not notice such behavior so easily.

Other, more obvious protection was done less frequently. Callbacks were corrupted, which made the game crash in odd ways. The European version changed languages randomly. Some of these actions break the game and others are just an annoyance to the player, but if the game is difficult or frustrating to play because of the failed crack, this can be more effective than breaking completely.

By making the game behave in as many odd ways as possible, we hoped to cause a lot of confusion. The pirates wouldn't know if the crack didn't work, whether they had just failed to apply the

"If the crackers know what they are looking for, they often don't even have to boot up the game to find the protection."

ger and the game takes time, and the more often we could force the cracker to do this, the more of their time we were wasting.

Perversely, though, the harder a crack is to make, the more fun it is for the cracker to make it. It's a challenge, and therefore fun. Paradox, the cracking group who produced the working patch for YOTD, even thanked the "Sony coders" who added such interesting protection to the game. The more difficult the crack, the more effort they will put into making it.

It's the same with the length of time it takes to produce. The longer that the game has been out without a crack, the more prestige there is in being the first to produce one. Again, this means more effort will be put into producing it.

Taking Action

Of course, all of this effort is worth nothing if the game doesn't do anything once a crack is detected, but this needs to be handled carefully. If the game just halts as soon as any modifica-

crack correctly, or if the disk had failed to burn correctly. The people who didn't play a lot of the game wouldn't notice that anything was wrong and claim that the crack worked. This happens more than you would think. A lot of people pirate more out of habit than anything else, booting up the game to have a look before moving on quickly. All of this would help to delay a complete crack from being made, because no one would be sure that it was required.

The Costs

Implementing all of this protection takes time and resources away from actually developing the game. For YOTD those costs were as follows:

Programmer time. One programmer was required for three to four weeks. The programmer spent this time adding the copy protection, integrating the anticrack protection into the game, and writing tools to mask the data and generate checksums. For about

six months prior to actually writing any code, some time was spent thinking of methods for protecting the game and what to do when a crack was detected. This was slightly less than two percent of the total programmer time budgeted for the game.

Game data preparation. The game data needed additional preparation before a disk could be burned. The game's WAD file had to be run through tools to generate checksums and mask data. This added about an hour to the burn cycle, making it about three hours long. The extra steps involved also made this process more prone to error, though this diminished over time as we became used to it and automated what we could.

Debugging. Any version of the game with protection included was very difficult to debug, as any software breakpoints would trigger the protection. Beyond a certain point, hardware breakpoints were turned off by the copy protection. This effectively meant that any debugging had to be done by the programmer who implemented the protection (me) on production versions of the game.

Testing. The protection was designed to produce effects almost indistinguishable from bugs, so testing was also affected. If any false positives occurred in the protection, they could be reported incorrectly. For this reason a very thorough debugging plan was produced just for the protection. Every location that could trigger protection was listed, along with how long it would take to trigger, what the exact effect would be, and where you had to look to see the effect. Testers had to visit the locations, wait the required amount of time, and then look to see if the protection had been triggered. Having any of the protection give a false positive was obviously our biggest worry. Therefore all the protection was set up on a compile-time switch so that it could be turned off at any time if we weren't absolutely sure that the protection was reliable (and believe me, there were a few moments when it didn't seem to be).

After the Crack

In the end, rather than trying to remove all the checksum code, the crackers simply found a way to bypass it. I'm not exactly sure how, but I know YOTD was vulnerable because the copy protection was only run once, at boot time. I assume the crack bypassed the copy protection and then restored the data to its original state. Any checksums performed after this point would not find any alterations (and any checksums before this were removed by the crackers).

While the protection on YOTD was reasonably effective, there were definitely things that we could have done better. If we had been able to check the data on the disk and run multiple copy protection checks, then it would have been a lot more difficult for the crackers. As I mentioned at the beginning of this article, there were practical reasons why these approaches could not be applied to YOTD. Maybe if the protection had been integrated into the game earlier, these difficulties could have been overcome.

Also, too much of the game could be played with a partial crack. This was a balancing act, though. If the protection had kicked in faster, perhaps the crackers would have realized sooner that they hadn't been successful with the first crack. But in the

end, we were perhaps a little too cautious. We could have reduced the amount of the game that could be played with an incomplete crack.

What We Learned

Were all our efforts worth it? Yes. While the effects of crack protection against piracy are extremely difficult to measure, we certainly caused a great deal of confusion. Until the crack came out, YOTD was the most talked about game on the copying forums. People wasted disks, blamed the cracking teams, and claimed that the cracks that didn't work were O.K., just because they hadn't seen anything go wrong. People were saying nasty things about Insomniac and Sony because they couldn't "back up" the game. Some people even thought it was funny when the fairy character, who normally offers players helpful advice, instead told them they were playing a modified game. There is also an effect on future piracy to consider: at the very least we made a few people think twice about buying a cheap copy of a game.

We've gained valuable knowledge about what works and what doesn't. Layering protection that doesn't kick in immediately is definitely a very effective protection. If nobody thinks a crack is required, they won't be working on one. Even when they do work on the crack, it takes them longer. The crackers apparently spent quite some time play-testing YOTD before they released the final crack, just to make sure they didn't get burned twice.

Unfortunately, the crack protection is weaker once the copy protection has been run. The cracker only needs to remove the code that runs the copy protection. Once it has been run, the original code can be restored, and the checksum will be correct. If this is only in one place, it is easier to attack. To combat this, the copy protection needs to be run as often as practical from independent copies of the code.

If there is space, put multiple copies of the game data on the disk. The cracker will have to find out which one is used or alter them all. Either way, you've slowed them down. An extension to this would be to actually use multiple copies of the data, either loading a random selection or loading using a pattern based on when the data is being loaded. If some of the copies are masked differently and some are never used, the cracker will have to find and alter them all to ensure that the crack is complete.

Even better than masking the data is compressing it, which offers many advantages over simple masking. The relationship between compressed and uncompressed data is much less obvious, the file sizes are different, and any cracked data has to be compressed or else it won't fit back on the disk. This means the cracker has to find out what compression was used, and if you customize the algorithm for your data, they may have to write a compression program just to be able to make the crack.

Looking back at the choices we made, we could have implemented multiple copy-protection checks throughout the course of the game. Unfortunately, this isn't always possible or practical, depending on the method of protection used (especially if minimizing load times is a primary concern). An alternative is to check the source data on the disk. Of course you can't check the entire disk, but all the executables can be checked, along with the table

of contents and boot information. This is something YOTD failed to do and is probably how it was cracked.

Reality Check

We may not be able to stop the pirates, but we can have enough of an impact to make pirating a much less attractive option. Given the choice of buying a game or waiting two to three months for a pirated version, a lot of pirates are going to start buying games. Or at least they'll buy their favorite ones.

There is also an advantage in numbers; the more games that add effective protection, the greater the benefit is for all games. Crackers have limited resources, and the longer that they're tied up on each game, even if it's only for a few weeks, the fewer cracks they can produce.

Games that implement just a standard copy protection scheme can be cracked in less than a day. Sometimes a tool is even available which does it in seconds. Any game that takes longer than this because of added protection will be put in line until the cracker has time to deal with it. The longer that line is, the longer it will take for any given game to be cracked. The trick is to keep your game from reaching the front of that line for as long as possible. ☺

FOR MORE INFORMATION

If you are interested in learning more about how the copying community works and how cracks are made, try looking at the cracking groups and forums. Here are a few starting points.

www.paradogs.com

www.cdrom-guide.com

www.gamecopyworld.com

www.megagames.com

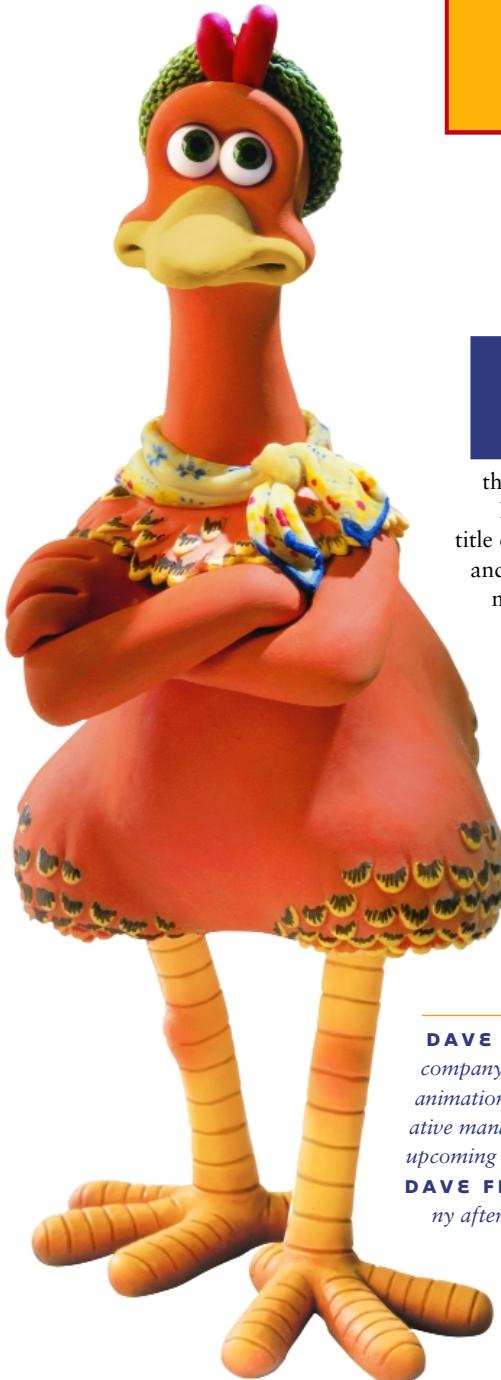
The following links are not to crackers but to "homebrew" programmers who make console demos. Still, techniques and tools made for the hobby scene always end up migrating to the crackers.

www.uic-spippolatori.com/psx/tute/faq.html

www.hitmen-console.org

<http://napalm.intelnet.com>

Blitz Games' CHICKEN RUN



By the end of 1999 it was becoming increasingly unviable for third-party publishers to continue to support the Nintendo 64. Like many independent developers, Blitz Games suffered because of this, but it was the cancellation of one of our N64 projects that indirectly led to our decision to approach the CHICKEN RUN games.

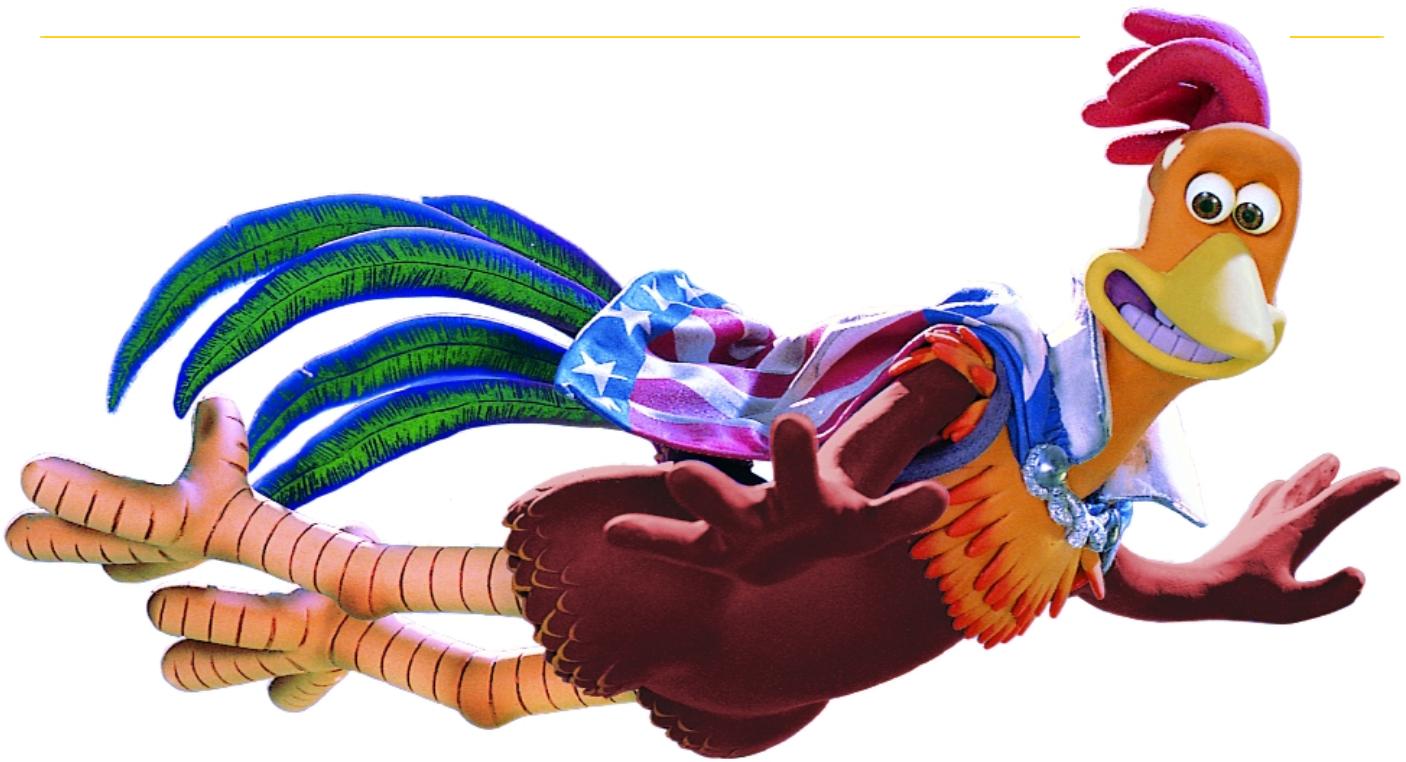
Earlier in the year, we'd had much success with an original Nintendo 64 title called GLOVER. Work was well under way on a sequel for both the N64 and Playstation when both projects were pulled by the publisher, leaving more than 20 people without a game to work on. This happens to many developers, of course, but Blitz never likes to put people out of work in a situation like this. The search was on for a new project to keep these two teams occupied and self-sustaining.

At around the same time, our chief executive and co-founder, Philip Oliver, was at an industry dinner in London and got talking (in the men's restroom, of all places) to an industry friend. He mentioned that he might have a very attractive project coming up that would be right up Blitz's alley if only we had the resources available. Philip was naturally very interested, and a meeting was soon set up where we learned that the interactive license in question was *Chicken Run*.

Dreamworks (which owns the *Chicken Run* license together with Aardman and Pathe) had been looking for an interactive

DAVE MANUEL | Dave has been with Blitz Games since 1997. He joined the company after doing a degree in graphic design and a master's degree in computer animation. He has worked on a range of Blitz titles, including GLOVER, and was creative manager for the CHICKEN RUN titles. He is currently a project manager for an upcoming Xbox title.

DAVE FLYNN | Dave has been with Blitz Games since 1997. He joined the company after spending numerous years writing and selling his own software while studying multimedia at university. He has industry experience spanning back to 1993 and has worked on a range of other Blitz Games titles and was a team leader on CHICKEN RUN. He is currently an artist for an upcoming Playstation 2 title.



licensee for some time but had so far drawn a blank. This was partly because they were keen to have a finished game for release for the Christmas 2000 holiday period in order to capitalize on the last major push for the Playstation 1, but that left less than a year in which to develop the game. We had to act fast and make some tough decisions. We had more than 20 people back at the office without any work, but we had the opportunity to venture into new territory for a developer — paying to become a licensee ourselves. After much deliberation, we picked up the full interactive rights for the *Chicken Run* property and set about putting the games together.

Although we knew we had the manpower to produce the three main console versions of the game (on Playstation, PC, and Dreamcast), we decided to subcontract some of the other versions that were produced. We realized that, as well as supporting a console-style game, the PC audience would also be interested in a multimedia-type package. So we approached Activision (which had just sold record numbers of the *TOY STORY ACTIVITY CENTER*) and a developer called Absolute (which had worked with Aardman, the studio behind both *Chicken Run* and the *Wallace and Gromit* films, previously on several *Wallace and Gromit* multimedia packages), and contracted them both to produce the *CHICKEN RUN* PC FUN PACK in time for the U.S. and U.K. movie release date. We also decided that a Game Boy Color version of the game could work well, so we prepared detailed design documents and approached specialist Game Boy Color programmers to follow our design. The finished game, although different from the main console versions, was ready for release at the same time and published by THQ.

If we were going to hit the pre-Thanksgiving/Christmas release slot with the three versions that we were developing in-house, we needed to put a pretty aggressive schedule in place, which is what we did. The GLOVER Playstation team got to work on the Playstation version of *CHICKEN RUN*, with the plan to convert to PC and Dreamcast later in the project. Meanwhile, the GLOVER 2 N64 team got stuck into the next-generation versions, planning to produce a Playstation 2 title initially, followed by Xbox and Gamecube versions.

That's not the end of the story, though. Although we had the rights to develop the games, we still needed a publisher on board to get the finished titles to market. So while the teams put together a detailed schedule and got started on the work, the senior management team began to make their way around the major publishers with the first playable demo in order to get a deal signed up. We were always confident that the *Chicken Run* movie would be a smash hit because we'd seen plenty of Aardman's work before, such as the *Wallace and Gromit* series, which is very popular in the U.K. Many of the American



GAME DATA

PUBLISHER: Eidos Interactive

FULL-TIME DEVELOPERS: 24

BUDGET: Around \$1 million plus the cost of the license

LENGTH OF DEVELOPMENT: 9 months

RELEASE DATE: November 2000

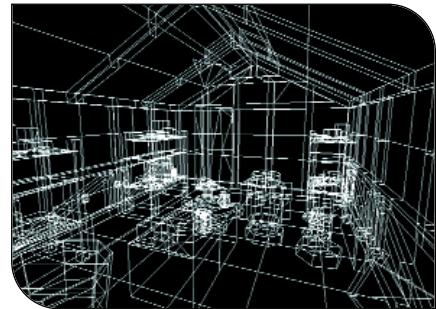
PLATFORMS: Playstation, Dreamcast, PC (×2), Game Boy Color

HARDWARE USED: 450MHz Pentium IIIs with 64MB RAM, respective development kits

SOFTWARE USED: 3D Studio Max, Photoshop 5.0, Painter, Visual SourceSafe, InstallShield, Visual C++, Bink

NOTABLE TECHNOLOGIES: Jobe: in-house 3D texturing package; Egg: in-house world-creation package, written specifically for the *CHICKEN RUN* games.

Chicken Run images™ & © 2000 Dreamworks LLC, Aardman Chicken Run Ltd. and Pathe Image. *CHICKEN RUN* games designed, developed and licensed by Blitz Games Ltd. *CHICKEN RUN* games published under license by Eidos Interactive, THQ, and Activision.



The interior of one of the chickens' work huts in various stages of modeling and texturing.

publishers were yet to be convinced, though, and it initially proved to be an uphill struggle. A deal was eventually signed with Eidos on the night before E3 2000, and everything was in place.

What Went Right

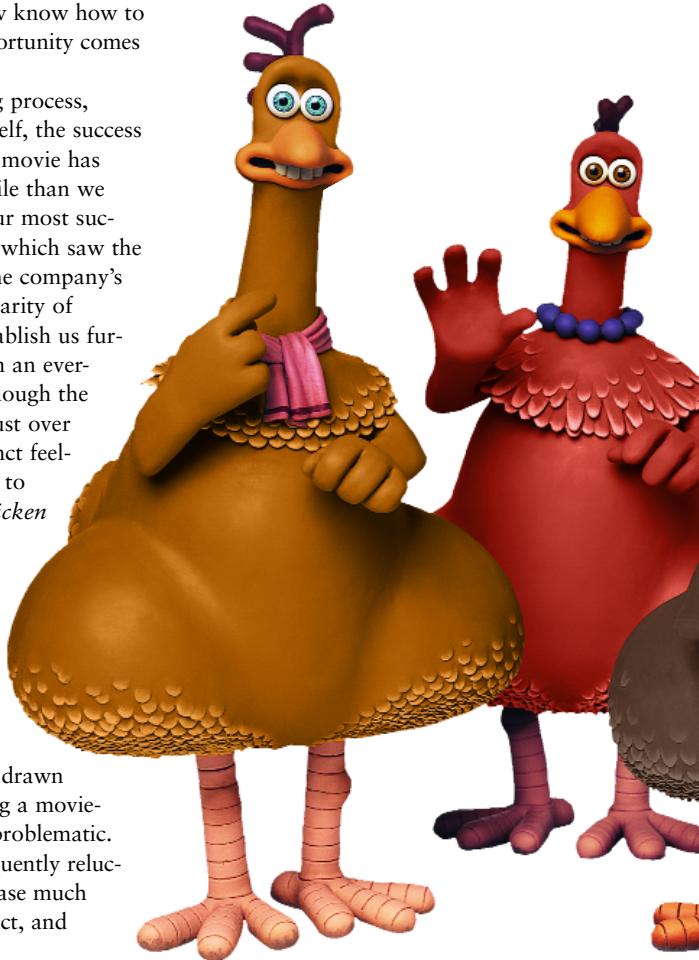
1 • Becoming a *Chicken Run* licensee. One of the most successful aspects of the project was the fact that we ourselves were a licensee of the property, rather than just a third party commissioned by a publisher. We have a five-year deal for the interactive rights for the property, and this gave us a much higher degree of control than on any previous project. We obviously had consultation and approval processes with Dreamworks, Aardman, and our publisher Eidos to consider, but the project-management freedom which being a licensee gave us was liberating.

The whole deal was obviously a very risky move, as we carried the license fee and the whole of the development costs ourselves, but in the end the risk paid off. We were one of the first independent developers to pay for a license option ourselves, and we've gained a lot of respect within the industry for going down this route. It was hard work trying to sell the game to publishers in this way, but we've learned a

lot from the process and now know how to approach it better if the opportunity comes up again.

In addition to the learning process, which was worthwhile in itself, the success and public awareness of the movie has given us a much higher profile than we had before. Last year was our most successful year to date and one which saw the biggest lineup of games in the company's history, but the global popularity of *Chicken Run* has helped establish us further and raised our profile in an ever-growing range of areas. Although the company has been around just over 10 years now, there's a distinct feeling that we're really starting to make our name. Being a *Chicken Run* licensee has helped strengthen that position and improve our reputation for creating quality games.

2 • Getting great support. If you're a longtime reader of these Postmortems, you may have drawn the conclusion that producing a movie-licensed game can be really problematic. The film's producers are frequently reluctant or simply unable to release much information about the product, and



while the game needs to be tied into the movie closely, it's often impossible to get much information out of the studio. Often, little more than a preliminary movie script is offered to the development team on which to base the look, feel, and gameplay of the games, but in our case we had no end of help from Dreamworks and in particular Aardman themselves.

The team at Aardman was very helpful right through the project and provided us with a wealth of source materials on which to base our designs for the virtual chicken farm. As Aardman's base is a mere hour's drive away from ours, they showed us the sets for the movie down at their studios, let us see scripts and storyboards, and also let us have full style-guide reference materials and snippets from the finished film to use in FMV sequences throughout the game.

The great thing was that because the movie is an animated feature, the film's production work took even longer than producing a game usually does. By the time we came on board, a lot of the materials and footage was already available. Our team was also able to see rough cuts of the movie well ahead of release. This level of support ensured that everyone working on the project had a good view of how the finished movie product would eventually appear in cinemas worldwide.

3 • Making the right game-play decisions.

It's always tricky

to take a linear movie story line and make it into an interesting interactive experience. It was especially tough in this case, as the film is essentially about failure almost right up to the last scene. We realized that to make the game enjoyable there had to be an element of player success much earlier on in the game, so it was important that we include plenty of opportunities for at least some of the chickens to escape at various points. To do this, we decided to include a number of amusing sub-games that would also add replay value to the

game, whereby the player could free a group of chickens in one go after building a contraption of some sort.

Complete contraptions. It was these sorts of contraptions that we also felt were key to the movie's (and therefore the game's), distinctive humor. The array of weird and wonderful devices that the chickens create and use in the movie led us to employ a simple but effective find-and-collect element to the main gameplay, and we were happy at how well this worked.

Chicken Gear Solid. The other major gameplay decision we made early on that was vital in re-creating the real feel of the movie was to include stealth elements. The film's *Great Escape* overtones naturally led us to implement the same idea, and pay homage to the likes of *METAL GEAR SOLID* in the game. We used several MGS-style elements (such as the radar style and characters creeping along walls) in the early designs, and these helped give the characters a slightly more heroic edge as they pitted themselves against guard dogs, searchlights, and the evil Tweedys.

Although a few reviewers actually accused us of stealing *METAL GEAR*'s idea, most realized we were poking gentle fun at a genre in the same way that Aardman uses subtle visual in-jokes in its movies.

The main plus point in our game-



Ginger and Babs discuss their next move in one of the many cut sequences.

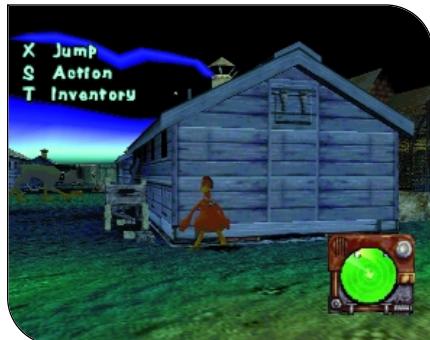
play decision-making was the fact that although we were guided strongly by the movie's visual appeal (discussed in the following section), we didn't make the mistake of following the proceedings so slavishly as to cripple the development process. In the end, the game was praised for being similar to the film while still being imaginative and creative.

4 • Faithfully re-creating the atmosphere of the movie.

Aardman is known for its perfectionist approach to its work and is rightly very keen to make sure that any use of its characters, environments, or story lines is treated sympathetically. Luckily for them, Blitz is a company that recognizes the value of strong intellectual property, and we have much experience taking established characters and giving them an interactive spin. Above all, we were keen to make sure that the CHICKEN RUN games were something much more than your average movie tie-in and that they extended the experience and the longevity of the film.

Character-building. As we've said already, the movie has a very distinct look and feel, and not just because the main characters are all made out of plasticene. It is these characters, though, that are always the strongest element in any Aardman production, and we were keen to make sure that anyone who had seen the movie would instantly recognize the





Stealth and cunning play a major role in the CHICKEN RUN titles. Characters sneak around the huts at night in METAL GEAR SOLID style (top left), avoid the spotlights and Mr. Tweedy's flashlight in the farmyard (right), or negotiate a tractor puzzle in the toolshed (bottom left).

look and personality of any of the characters in the game. This was a difficult feat, because lip-synching and facial animations, which would have achieved this easily, are near-impossible on the Playstation. We therefore worked closely with Aardman to create realistic body movements and gestures that would accurately convey the personality of each of the characters. Many of the movements we wanted the chickens to do in the game were things that they'd never had to do in the film, but the Aardman animators were again on hand to advise our own animation teams with tips on how the characters were originally brought to life.

We did a detailed study of each chicken, but in some cases we only had a few hundred polygons for each one, so we carefully combined a mix of body gestures with animated tex-

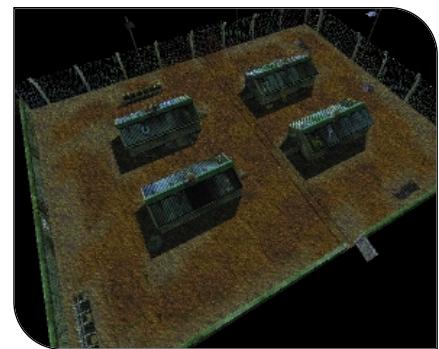
tures to create just the right feel. The plasticine look of the characters was then achieved by our expert texture artists. They worked with subtle but highly detailed bitmaps, which were combined with some advanced lighting techniques coded into the game.

Moonlight serenade. Another challenge the team faced was that a large chunk of the movie takes place at night in the moonlit farmyard, and these moody conditions are tricky to translate into an interesting gaming environment. The dark and hostile environment we saw in the movie was very distinctive, and the chickens are constantly on the lookout for guard dogs, or the Tweedys as they plot their escape. We knew we could introduce that stealthy feel to the gameplay as we've already discussed, but we also needed to create a visual feel that supported it.



Once again, it was Aardman's help that enabled us to do this so convincingly. Along with their original plans of the farmyard, they also let us have source stills and production shots of the set. From these we were able to create the final layout of the gaming environment. Each building was located exactly where it was in the film, but we also gave each one a detailed interior and populated it with collectibles, chickens, and plenty of other environmental details.

Lighting the way. Lighting was another key element of the environmental design. The blue-tinted moonlit sections worked really well in the end, despite some initial confusion as to how the game's lighting effects should be implemented. For much of the level lighting we used an in-house tool that enabled us to have low-level control over all aspects of texturing and lighting on a per-polygon basis. Each of the meshes was painted with Gouraud light, which let us create atmospheric areas of deep shadow as well as the piercing light of the farm's spotlights. We added animated scenic models to many scenes, as well as a host of other smaller touches, such as flickering candles and smoke drifting from



The level layouts are very faithful to the movie, whether it's the chicken coop itself (bottom right) or the farmyard outside the wire (left). It's all tied together with cutscenes and conversations with the game's key characters (top right).

stoves, in order to convey the Aardman feel authentically.

5. Including voices. The inclusion of character voices was a bone of contention for some time during the course of the project. Many on the team were unconvinced at first that voices could have such a major impact on the finished product, but our development director, Andrew Oliver, insisted they be implemented. In retrospect, we all admit that he was right to force our hand, because the character voices ended up providing that final piece of character expression that would otherwise have been lacking.

As we mentioned in the previous point, it is the characters in Aardman's work that are its strength, and it's their humorous phrases and expressions that really complete the picture. Once it was finally agreed that we would include voices in the game, the work began in earnest to secure voice talent to record the extensive script we'd drawn up. We managed to secure a few of the original actors, but for many of the characters we began a long search for sound-alikes, which then had to be approved before any recording

could begin. There was much confusion over whether or not we would be able to use all of the key character voice talent from the film in the game, and we went to great lengths to try to secure it. We ended up having to use some sound-alikes, but once again we learned some valuable lessons in how to control this type of input into a game.

All our effort was well worthwhile, and the voices add that finishing touch to the final game. The voices give the characters more life and personality than simple text dialogue would have provided.

What Went Wrong

1. Attempting too many versions. We'd been very excited about the CHICKEN RUN project, and after assigning so much of the company's money to buying the rights, we were naturally keen to make the most of it. We initially planned to produce a number of titles on a range of platforms. The idea was to have Playstation, PC, and Dreamcast versions ready for release in time for Christmas 2000, then to follow them in 2001 with Playstation 2, Xbox, and eventually Gamecube versions.

Unfortunately, this wasn't to be the case.

As we mentioned earlier, we had trouble at the start of the project convincing many publishers that the movie would be a success. Even though we were eventually proven right, we then had trouble convincing them that interest in the property would extend beyond Christmas 2000. We had full designs in place for the next-generation versions and had a team working on them, but it soon became apparent that the work on these games was in danger of jeopardizing the earlier-release titles. As it was looking increasingly unlikely that we could secure publisher support for the next-generation versions anyway, we decided that work should stop on these games and that the two teams should be merged in order to guarantee the production of the Playstation, PC, and Dreamcast versions.

It's a valuable lesson to have learned, but perhaps it was inevitable that having paid out for the interactive rights we would try to throw as many people into production on as many different versions as possible. As we said in the first part of the "What Went Right" section, what we have learned from CHICKEN RUN on all levels is valid. If we are ever in the same

position again, we'll be much more realistic with our goals of what we can do with a license opportunity of our own.

2. Having too short a schedule.

The major problem throughout the project was time, or rather a lack of it. We had nine months in which to produce all three versions of the game, and by most standards these days that is an extremely short project. We didn't want to skimp on any aspects of the game, but at the time the team merger was first suggested it was becoming more and more obvious that something had to give. Combining the staff helped enormously, but at the same time we had to make some hard decisions about exactly what would remain in the game and what might have to go.

Removing sections of a game midway through the development process was never going to be a popular move, but when the game was assessed in April it was decided that a large section set in the Tweedys' pie machine would have to be cut. This obviously caused a few ructions with the modelers that had started work on it. Nevertheless, this

section was chosen for a number of reasons, not the least of which was that it had the least amount of already completed work in place.

Once these decisions had been made, though, the team was able to carry on with the project with a much greater sense that the whole package was achievable. It was still a massive amount of hard work and determination from the team that actually brought the games to completion on time. If there's one thing we'll take from this project on to the next one, it's that we need more than nine months.

3. Re-creating the scale of the movie.

As we said in the "What Went Right" section, we tried to re-create the whole movie's environment, along with some new areas, as faithfully as possible. Still, the scale of the farmyard is something that was never going to be easy to get across. We were keen to make sure that as players guided a small chicken across the coop and the farmyard, they got a real

sense of how large and foreboding their surroundings were. As we explained, we achieved a lot of this with atmospheric visual touches, but we also wanted to let players explore the entire farmyard if they wanted to in order to give them the sense of scale of the place.

The PlayStation's limited memory caused us to struggle with displaying the farm in its entirety, so we had to split it into a number of separate sections and stream them in from the CD one at a time. This process worked really well and enabled us to create a feeling of seamless exploration across the entire farm. It also permit-



Nick and Fletcher, the wily Cockney rats, tackle the infamous pie machine.

ted a much higher polygon count for each hut and room interior that was displayed.

4. Merging two teams into one.

When two groups of development staff, one of which has already spent a few months working on a project, are brought together to complete the project, it's inevitable that there'll be some tension. Not surprisingly, this was the case with CHICKEN RUN. As we've already discussed, merging the teams was essential if all the planned Thanksgiving 2000 releases were to be delivered on time. However, when the moves came, not everyone on the teams was completely happy, and morale problems began to surface.

Part of the problem was that the two team leaders had distinctly different managerial styles, and as a consequence the creatives in their teams were used to working in different ways. The nature of the short schedule also meant that this transition period was considerably more rushed than it would have been if it had occurred during a longer project.

Once again, we learned some valuable lessons from this team-merging, not the least of which was that it's better to sort out the team structure before simply moving desks about the building. Many of the staff were unsure of the new structure or how it would affect them and their work, so at first productivity suffered slightly. However, with some determined reorganizing and fresh planning of the task at hand we were able to pull off what at first seemed impossible — finishing all three versions of the game on time.





Rocky dashes into the egg store to avoid the evil Mrs. Tweedy (left). Ginger's safer inside one of the huts (bottom right) than in the glare of a flashlight (top right).

It's easy to say that you've learned from your mistakes, but in this case we have already implemented a new approach to staffing. We are now approaching the games that we currently have in development in a much more flexible way. Team structure and identity are just as important as ever, but with new systems in place we can now apply several teams to one game project as they're needed without too much upheaval.

5 Starting testing and bug-fixing too late. CHICKEN RUN's short timescale affected us in many ways, but one factor that proved significant was that the development work on the Playstation version was scheduled to beyond alpha stage. This meant that a lot of the play-testing and then bug-fixing couldn't be started until much later than we would have liked.

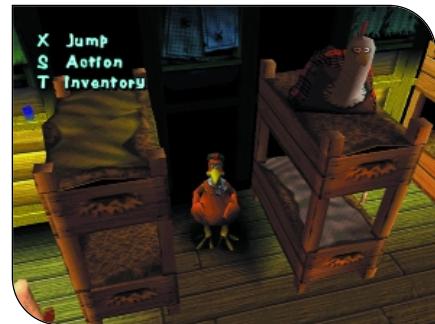
Play-testing always brings up a huge range of suggestions for improvement, and this was the case with CHICKEN RUN. Several ideas to make the game more playable were suggested, including modifying the flying-machine section at the end of the game. This was a significant element to alter, and we were reluctant at first to embark on

such a major change so late in the project. Eventually, the work was done, and it made the final version much better. In turn, however, it brought up a whole raft of new bugs to work through at a late stage.

We also encountered problems when we started to identify bugs. We were using an internal spreadsheet system for recording all bugs, but this wasn't compatible with the system our publisher was using. Because of the unusual nature of how this project worked (in that we were the licensee and were essentially just selling a finished product to Eidos), the whole bug-testing scenario was unclear for some time. If it had been possible for all parties to register bugs into the same system, then it might have been possible to smooth out the bug-recording and -fixing issues. In a project with an already very tight schedule, this would have been a big help toward lessening the growing tension in the final weeks of development.

Inspiration and Determination

The key factor throughout this project was the importance of time, and very major problems and changes ultimately



came about because we were working to a nine-month schedule. In retrospect, the team is pleased with the results, and although the game is perhaps a little short for a hardcore game player, the target audience has loved it. There are always problems in every project, and there are always things to learn as each game progresses. We had many more unknown quantities than usual when we took on CHICKEN RUN, and we all recognize that the project could so easily have failed. The reason it didn't was a combination of hard work and sheer determination, enabling us to win through. In addition, the whole *Chicken Run* property is one that really excited everyone here at Blitz, and the chance to work with some of the world's best animators was something that kept a lot of the team going through even the hardest times. Becoming a licensee for a property such as *Chicken Run* was an interesting new move for us, but one which was ultimately a great success, leading us to consider similar interactive licensing opportunities for the future. We consider ourselves very lucky that we were able to have learned some valuable lessons while still managing to put out a well-received and highly playable game on time for three different platforms. ☺

Software Patents Should Be Abolished

"He who receives an idea from me, receives instruction himself without lessening mine; as he who lights his taper at mine, receives light without darkening me." - Thomas Jefferson

Whether software patents are ethically wrong or not is a matter of philosophy. However, even if you don't believe patenting software is wrong from an ethical standpoint, you must agree that if a thing is not working as intended, and if it is in fact operating to the contrary of what it's intended to do, then that thing must either be fixed or thrown out.

We strongly believe software patents are unethical, but they also fail miserably to work as intended. Worse yet, they are detrimental to the game industry and to the software industry in general. We fear software patents will spell the end of freedom and innovation in game programming if we continue on our current course.

We hope that after reading this short article and following up with the references yourself, you will come to the same conclusion we have, and will join us in trying to dismantle the incredibly broken patent system.

Intent vs. Reality

Software patents do not work as intended on a number of levels. Patents — limited-time exclusive rights to an invention — are allowed under the U.S. Constitution as long as they "promote the progress of science and useful arts." Unfortunately, software patents do not accomplish this goal in practice, and hard evidence is emerging that they actually hinder progress for both the patent holders and for others in the industry.

James Bessen, from an organization called Research on Innovation, and Eric Maskin, a Harvard and MIT economics professor, have written a paper called "Sequential Innovation, Patents, and Imitation." In this paper, Bessen and Maskin look at the various rationales and conventional wisdom for patents, and build economic models for these theories. They then compare the theories and models with data from the computer industry. The software industry is a good empirical test case for patents, because software

patents were not legal until the early 1980s. So, the authors look at R&D spending and other metrics from the time before software patents until after they're well established.

Their results are not surprising to any programmer unlucky enough to have dealt with software patents: Software patents did not spur progress and competition in the computer industry. The industry did not experience a wave of innovation after software intellectual property law was strengthened, as defenders of software patents have implied. In fact, Bessen and Maskin found that R&D has stagnated relative to pre-software-patent times, especially among the companies that patented most.

This is vitally important research. The patent debate is rife with assertions and assumptions, but here is some actual data showing that software patents have had the opposite of their intended effect. As David Brotz, principal scientist at Adobe, testified, "It is clear to me that the constitutional mandate to promote progress in the useful arts is not served by the issuance of patents on software."

Given this data, even if you don't agree with us that software patents are inherently unethical (the topic for another Soapbox), as a logical person you still must conclude that they need drastic fixing or elimination. Unfortunately, laws are not changed by the publication of a research paper, regardless of how important its results are. Laws are changed by people who have a vested interest in an issue lobbying Congress. That's why it's important that you, as a game developer, learn more about this issue and become active.

Here are a few more brief reasons why software patents don't work:

"Nonobvious" to whom? The vast majority of software-patent criticism focuses on absurdly obvious algorithms and techniques being granted patents. A lot of these are quite hilarious in how they violate the U.S. Patent and Trademark Office's own criterion of being "nonobvious to a person having ordinary skill in the art." However, there's an incredibly serious side to these anecdotes. Patents which will affect your ability to develop games are being granted every day by underpaid and technically unsavvy



continued from page 64

patent examiners. Do you really want the future of your company or career to be decided by an examiner who has only eight hours to review a patent application?

Patents don't protect small inventors. Another myth is that patents protect small inventors from being crushed by large companies. The numbers do not bear this out. Applying for a patent is expensive, and prosecuting or defending a patent is astronomically expensive; Stanford law professor Lawrence Lessig says it takes \$1.2 million on average to litigate a patent.

Bad resource allocation. As Lessig, a lawyer himself, says, “Awarding [obvious] patents . . . siphons off resources from technologists to lawyers — from people making real products to people applying for regulatory privilege and protection.” Brotz said, “An industry that still generates tremendous job growth through the start ups of two guys in a garage will not continue to grow when a room for a third person, a patent attorney, needs to be made in that garage.”

The references we've provided cover many more reasons software patents are a bad idea. The truth is, software patents do far more harm than good, and they are forming a noose around our industry as you read this. Please read up on the issue and do what you can to help eliminate software patents. 

FOR MORE INFORMATION

- The League for Programming Freedom
<http://lpf.ai.mit.edu/Patents/patents.html>
<http://lpf.ai.mit.edu/Patents/against-software-patents.html>
- A pro-patent rebuttal of the LPF essay
www.heckel.org/Heckel/ACM%20Paper/acmpaper.htm
- Bessen and Maskin's paper
www.researchoninnovation.org/patent.pdf
- The IGDA is forming a Software Patents Committee
www.igda.org

CHRIS HECKER | Chris (checker@d6.com) is editor-at-large of Game Developer. **CASEY MURATORI** | Casey (cmu@funktroll.com) is the lead developer on *Granny*, RAD Game Tools' licensable character animation system and 3D toolkit.