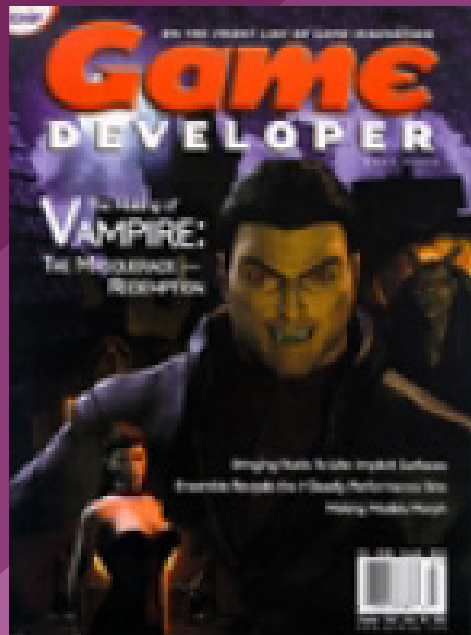




GAME DEVELOPER MAGAZINE

JULY 2000



GAME PLAN

LETTER FROM THE EDITOR

Welcome Back, K.C. Munchkin

I've yet to meet someone who doesn't have a strong opinion when it comes to the copyright cage match pitting Hasbro against eGames, Webfoot, MVP Software, and Xtreme Games. Not surprisingly, many small developers have been vocal in their support of the defendants.

If you're rooting for the underdogs in this case, don't blame me for saying this, but I think Hasbro will win.

I don't think that either side is inherently evil. Hasbro claims that the defendants "blatantly copied" Hasbro's CENTIPEDE, TETRIS, MISSILE COMMAND, and ASTEROIDS, among others. The defendants claim they improved on the original games to the extent that they've developed entirely new games. This case once again calls into question the extent to which you must evolve a game concept in order for it to be legit.

Software copyright law leaves judges a lot of room for interpretation. A judge looking at the Hasbro case would probably focus on three questions to determine whether copyrights have been violated:

1. Did the defendants add something of value to the original games so as to transform them into something new and different?
2. How much of the original work was taken by the defendants? What parts of the games in question were copied from the original? It doesn't have to be a lot, either — copying even a small portion of a game may be illegal if the court determines that it's taken from the "heart" of the original.
3. What's the effect on the potential market for the original game? A judge would consider whether the games in question deprived Hasbro of income or undermined a new or potential market for the copyrighted game.

Given this litmus test of the defendants' games, the judge will probably examine first question most deeply. The prosecution will have to prove that not enough value was added to the games in question so as to make them derivatives of the originals.

Hasbro will also cite the legal precedent set by *Atari Inc. v. North American Philips Consumer Electronics Corp.*, 672 F.2d 607 (7th Cir. 1982). In this case, Atari took

Philips to court for releasing K.C. MUNCHKIN, which Atari said was too similar to PAC-MAN. Initially Philips was victorious in court, but ultimately it lost on an appeal. The Seventh Circuit ruled that K.C. MUNCHKIN violated Atari's copyright, stating that the audio-visual screen display copyright had been violated, regardless of the originality of the source code.

Some who support eGames and the rest of the defendants point to the *Lotus v. Borland* case of 1995, in which Borland was allowed to copy the look of the Lotus 1-2-3 menu structure in its own spreadsheet, Quattro Pro. The court stated in the ruling that a menu command hierarchy was a "method of operation," which was exempted from copyright law. I don't think that *Lotus v. Borland* will be a strong enough precedent to save the defendants, however. A menu interface just is one small part of an application not at the heart of the product, and Hasbro is arguing that the defendants went beyond that.

Whether or not Hasbro is victorious, the company has reopened a can of worms and it's going to be up to a judge — possibly someone with little or no game play experience — to decide where the line is drawn between innovative and derivative game design. This case is yet another indication that our industry is big business, and how risky it is becoming to be an independent developer. And maybe that's what this case is really about: a big company flexing its muscles to frighten away upstarts.

Adios Mel, Welcome Lisa. Our Artist's View columnist, Mel Guymon, has decided to move on from the pages of *Game Developer*, and we wish him all the best. In his stead, we welcome Lisa Washburn of Vector Graphics, who will be sitting in for a couple of issues as our guest columnist. Check out her column on page 19.



Let us know what you think. Send e-mail to gdmag@cmp.com, or write to Game Developer, 600 Harrison St., San Francisco, CA 94107

ON THE FRONT LINE OF GAME INNOVATION
**Game
DEVELOPER**

600 Harrison Street, San Francisco, CA 94107
t: 415.905.2200 f: 415.905.2228 w: www.gdmag.com

Publisher
Jennifer Pahlka jen@mfgame.com

EDITORIAL

Editorial Director
Alex Dunne adunne@sirius.com
Managing Editor
Kimberley Van Hooser kvanhoos@sirius.com
Departments Editor
Jennifer Olsen jolsen@sirius.com

News & Products Editor
Daniel Huebner dan@mfgame.com

Art Director
Laura Pool lpool@cmp.com

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

Contributing Editors
Jeff Lander jeffl@darwin3d.com
Lisa Washburn article@vector.org

Advisory Board
Hal Barwood LucasArts
Noah Falstein The Inspiracy
Brian Hook Verant Interactive
Susan Lee-Merrow Lucas Learning
Mark Miller Group Process Consulting
Paul Steed id Software
Dan Teven Teven Consulting
Rob Wyatt Microsoft

ADVERTISING SALES

National Sales Manager
Jennifer Orvik orvik@cmp.com t: 415.905.2156
Account Executive, Silicon Valley, Western Region & Asia
Mike Colligan mcolligan@cmp.com t: 415.356.3486
Account Executive, Northern California
Susan Kirby skirby@cmp.com t: 415.356.3406
Account Executive, Eastern Region & Europe
Afton Thatcher athatcher@cmp.com t: 415.905.2323
Sales Associate/Recruitment
Morgan Browning mbrowning@cmp.com t: 415.905.2788


ADVERTISING PRODUCTION

Senior Vice President/Production Andrew A. Mickus
Advertising Production Coordinator Kevin Chanel
Reprints Stella Valdez t: 916.983.6971

MILLER FREEMAN GAME GROUP MARKETING

Marketing Manager Susan McDonald
Product Marketing Manager Darrielle Sadle
Field Marketing Manager Jennifer McLean
Marketing Coordinator Scott Lyon

CIRCULATION


Game Developer magazine is BPA approved
Vice President/Circulation Jerry M. Okabe
Assistant Circulation Director Kathy Henry
Circulation Manager Stephanie Blake
Circulation Assistant Kausha Jackson-Crain
Newsstand Analyst Pam Santoro

INTERNATIONAL LICENSING INFORMATION

Robert J. Abramson and Associates Inc.
t: 914.723.4700 f: 914.723.4722
e: abramson@prodigy.com

CORPORATE

President & CEO Gary Marshall
COO/Corp. President, Business Tech & Channel John Russell
President, Business Technology Group Adam Marder
President, Specialized Technology Group Regina Ridley
President, Channel Group Pam Watkins
President, Electronics Group Steve Weitzner
General Counsel Sandra L. Grayson
Vice President, Creative Technologies Johanna Kleppe
General Manager, CMP Game Media Group Greg Kerwin





FRONT LINE TOOLS

WHAT'S NEW IN THE WORLD OF GAME DEVELOPMENT | *daniel huebner*

NVIDIA INTRODUCES GEFORCE 2 GTS



Nvidia is working overtime to maintain its lead in the graphics processor race with the release of the new GeForce 2 GTS processor. At the heart of Nvidia's new chip is the Nvidia Shading Rasterizer, a new technology that enables advanced per-pixel shading. The technology has the ability to process seven pixel operations in a single pass simultaneously

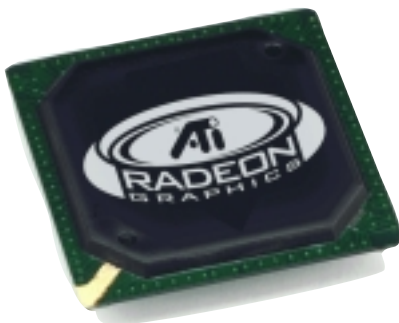
on each of the four pixel pipelines. The Nvidia Shading Rasterizer also allows for per-pixel control of color, shadow, light, and other visual components. The GeForce 2 GTS delivers up to 1.6 gigatexels per second and supports up to 128MB of double-data-rate frame buffer memory. The chip also includes a new high-definition video processor.



GEFORCE 2 GTS | [Nvidia](#) | [www.nvidia.com](#)

ATI LAUNCHES RADEON

No longer content with cleaning up in the laptop and Macintosh markets, ATI is shooting for the top with its most powerful graphics processor ever. Dubbed Radeon 256, the chip is built on a trio of new technologies: the Charisma Engine for geometry processing, Pixel Tapestry for rendering, and Video Immersion for digital video. Radeon 256 also boasts support for animation features such as advanced vertex skinning and keyframe interpolation. The chip delivers 1.5 gigatexels per second, and includes support for twin Radeon 256 chips on a single card using ATI's MAXX multi-ASIC technology. Radeon supports up to 128MB of double-data-rate memory at 200MHz and uses Hyper Z technology to boost effective memory bandwidth by 20 percent, enabling the Radeon 256 to access 8GB per second of effective memory bandwidth. The Radeon 256 is scheduled to begin appearing on boards sometime this summer.



RADEON 256 | [ATI Technologies](#) | [www.ati.com](#)

NEW OPENML STANDARD

A collective of industry leaders is lending its expertise to the creation of a standard API for graphics, video, and audio media devices. The standard, called OpenML, offers a standard technique for input and output of audio and visual data in an effort to increase application portability across operating systems and CPU architectures.

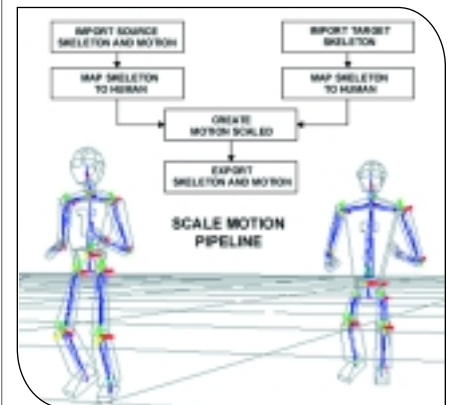


The API also offers extensions to OpenGL to handle compositing of multiple graphics and video streams. The Khronos Group, whose members come from companies including 3dfx, Discreet, ATI, SGI, and IBM, is leading the development.

OPENML | [The Khronos Group](#) | [www.khronos.org](#)

TESTAROSSA'S VROOM

Testarossa has released version 1.1 of Vroom, an application that uses an advanced biomechanical model to define behavioral rules of human motion and automatically perform complex operations based on these rules. The system allows multiple users to work on different assets at the same time, so that animation can commence before a model or skeleton is finished. Vroom supports retargeting animation data from one skeleton to another, updating motions to a new skeleton hierarchy, normalizing the data to a common reference pose, aligning and rotating motions along one direction, creating motion loops and transitions, creating mirror motions, and converting between file formats. Vroom is a stand-alone program for Windows NT/2000 and IRIX with prices starting at \$3,250.



VROOM 1.1 | [Testarossa](#) | [www.toolsinmotion.com](#)

STEINBERG ADDS CUBASE VST 5 TO THE MIX

Steinberg has updated its Cubase music production series, giving the system a complete graphic makeover for version 5.0 and adding new features to enhance usability and sound quality. Users of the top-of-the-line Cubase VST/32 can take advantage of 32-bit floating-point files for recording, output, and mixdown, as well as analog sound with Steinberg's True Tape process. Also new for this version are the Linear Time Base system for precise MIDI timing, a new FX and plug-in system, and the InWire Studio for online collaboration. Cubase is available for both PC and Macintosh platforms.



CUBASE VST 5.0 | [Steinberg](#) | [www.steinberg.net](#)



Take-Two Finds G.o.D. Gathering of Developers is promising to retain its developer-focused business operations despite becoming a wholly owned subsidiary of Take-Two Interactive. Under the terms of the deal, G.o.D. will operate autonomously and continue to handle its properties and games in North America. The arrangement gives Gathering better access to Take-Two's extensive European and North American distribution network while Take-Two gains Gathering's top-end PC and console game catalog. The two companies have been working together since Gathering's founding in 1998, when Take-Two provided part of the company's initial funding. That relationship expanded in 1999 when Take-Two took a minority stake in G.o.D. Take-Two has signed a five-year contract with Gathering CEO Mike Wilson and a three-year deal with president Harry Miller to keep them on in their current positions, and Gathering's management will stay in place to continue the development of the company's catalog. Financial terms of the acquisition were not disclosed.

Activision Cans Expert. Though the company is expecting strong revenue and earnings growth in its 2000 fiscal year, Activision is taking some aggressive restructuring steps. As part of its reorganization, Activision is folding budget games subsidiary Expert Software into its other value line, Head Games. The consolidation includes the cancellation of all of Expert's ongoing projects as well as the termination of its entire workforce and the closure of Expert's Miami headquarters. Activision is also looking to discontinue non-core product lines in order to focus its product range on titles with online or next-generation console potential. Activision expects approximately \$66 million in pretax charges related to the restructuring. The company is taking defensive measures to ward off any unwanted takeover bids as it undergoes this reorganization by enacting a stockholder rights measure. The plan grants shareholders additional stock or compensation at a rate roughly double their current holdings, but with the new rights kicking in only if someone makes a move to acquire 15 percent or more of the company. Those already holding stakes of better than 1.5 percent do not trigger the rights plan.




Take-Two's acquisition of G.o.D. will add the upcoming three-game BLAIR WITCH series to Take-Two's bottom line.


Sega Promotions. Sega of America has promoted several key executives as it moves into a critical stage of its Dreamcast strategy. Chief among the changes is Peter Moore's promotion to the position of president and COO. Moore had previously served Sega as vice president of marketing. In his new role, Moore is responsible for directing Sega's console and online gaming business in North America. Sega also promoted Shinobu Toyoda, one-time president of Sega's PC games division, to the position of executive vice president of content strategy in charge of Sega's game lineup. Chris Gilbert moved to the role of executive vice president of sales, marketing, and operations, while Neal Robison has been promoted to vice president of third-party licensing.

Nintendo Under Investigation. The European Union Commission is looking into possible Nintendo price fixing. The EU's Competition Commission believes Nintendo and its European distribution partners, Linea GIG SpA, Itochu Corp., Concentra LDA, Bergsala AB, Nortec SA, CD-Contact Data GmbH, and John Menzies PLC, acted as a cartel to divide up the European market and stifle competition. The commission believes that the group's actions wiped out parallel trade between EU nations and resulted in product price differences of as much as 100 percent from one country to another. The commission can impose fines as high as 10 percent of the company's annual global turnover. Sega, meanwhile, is the focus of a probe by the United States International Trade Commission into whether memory chips used in the Sega Dreamcast console violate patents held by Rambus. Rambus asked the trade com-

mission to investigate in March when it filed an infringement suit naming Sega and Hitachi, asking that the trade commission block imports of the contested chips.

Chipmakers Can't Slow Down. The graphics acceleration landscape continues to change, as 3DLabs issued up to 3.69 million common shares to acquire Intergraph's Intense 3D accelerator division. 3DLabs will pay up to \$25 million in additional stock or cash if Intense 3D meets certain performance marks, and the company will continue to provide graphics accelerators for Intergraph's product range. 3DLabs considers the Intense 3D product line to be complementary with its own offerings and has no plans to discontinue products or reduce the workforce at Intense 3D's Huntsville, Ala., headquarters.

Meanwhile, long-time laptop graphics chipmaker Neomagic is following S3's lead by eliminating its PC chip business to focus on the emerging market in Internet appliances. The company is setting its sights on developing technologies for MPEG-4 video, broadband wireless communications, and Internet system integration. Organizational changes there include the elimination of two departments and addition layoffs resulting in a 35 percent reduction in Neomagic's workforce. 



UPCOMING EVENTS
CALENDAR

NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE
LOCATION TBD
Austin, Tex.
July 30–August 3, 2000
Cost: \$605 (member and student discounts available)
www.aaai.org

LINUX WORLD
SAN JOSE CONVENTION CENTER
San Jose, Calif.
August 14–17, 2000
Cost: \$25–\$795 (early-bird rates available)
www.linuxworldexpo.com

PRODUCT REVIEW



THE SKINNY ON NEW TOOLS



The basic Pro Audio 9 window look: toolbars across the top, the main global track display, and a stylish 3D mixing console on the bottom.

Cakewalk Pro Audio 9

by gene porfido

When I was 15 and learning the piano, I wanted to play the *Peanuts* theme “Linus and Lucy” with both hands, but I just didn’t have the chops. That song was the very first thing I recorded when I bought my first sequencer. I added the left and right hand melodies one track at a time to create the *Peanuts* song in all its glory. I felt like I had actually played it with my own two hands. The feeling was unforgettable — I could now play anything I could imagine. Then the first version of Studio Vision arrived where you could actually record four tracks of audio *right into the sequencer*. My first attempt at using this innovative software was experimental but everyone that was in the studio knew the possibilities. Just seeing that audio wave-

form on the same screen as those MIDI tracks boded great things to come.

Since then there have been many milestones in sequencer technology, each expanding what musicians and nonmusicians alike can accomplish with patience and know-how. Cakewalk’s Pro Audio has become one of the most popular sequencers on the market, and version 9 packs a lot of punch.

The Basics

Pro Audio 9 follows a long line of successful and popular versions of this PC MIDI and audio sequencer. It is

versatile and feature-packed to the hilt, and easily recognizes most of today’s common audio hardware and sound cards, with special support for Yamaha’s DSP factory and Sonorus’s STUDIO/O cards. Up to 128 virtual tracks can be recorded at 16, 18, 20, 22 and 24 bits, with sampling rates up to 96kHz. And thanks to Pro Audio 9’s new Wave Pipe technology you can use up to 128 real-time audio effects, like a vintage amp sim, four-band parametric EQ, and tons of delay-based effects.

Installing the software is simple enough, although getting it to work with my MIDI hardware was problematic. It took a couple of hours of configuring, reinstalling drivers, uninstalling drivers, querying tech support at Opcode (I use a Studio 64 XTC for my MIDI interface) and Cakewalk, and searching online resources before the software and hardware finally jelled, at which point it was easy to get up and running. This is really no fault of Cakewalk’s, but they could have included more in-depth help for a larger variety of setups. Anyone using a serious rig with four or more MIDI devices and an interface other than the sound card might not find the help they need.

Walking the Walk

The program interface is classic Windows, with an army of toolbars, key commands, and right-click pop-up dialog menus that can control virtually everything in Pro Audio 9. There are a number of views available for recording, editing and mixing. The main Track view is divided into the Track pane, where initial settings like Mute, Solo, Record-Enable and Track Number are incorporated, and the Clip pane, which shows the song in a horizontal timeline and includes MIDI and audio overviews. The Console view is your virtual mixing board, with faders, patch points for effects, and just about anything else you would expect a real mixing board to include. The Piano Roll view, resembling a 2D piano, is an easy view in which to edit note placement, velocity, and controller

AUTHOR’S BIO | Gene Porfido spends most of his waking hours buried under a pile of computers and musical instruments. When he manages to dig free, he can be found on his Harley searching San Francisco for a real East Coast-style pizza joint.

information. The Staff view shows your composition in musical notation, and is a great tool for printing and handing out sheet music to fellow musicians during a gig. There's also the Audio view, which displays audio tracks in their waveform to allow most audio editing and arranging. The List view lets you get down to very detailed and specific editing of most parameters. Each of these views provides vertical and horizontal zoom tools for quick adjustment. You can also save the views as layouts that reappear exactly as you set them up.

Pro Audio 9 also provides windows that perform special functions including tempo and time-signature changes, markers, SYSEX to control SYSEX (system exclusive) messages, Studio Ware (virtual templates of your MIDI gear), a large time readout called Big Time, a video import, and a lyric view. All of these different views and windows contribute to the power of the software and make it highly customizable to fit individual tastes.

Cutting the Cake

Recording MIDI and audio are extremely easy in Pro Audio 9. On each track you can select input/output source, instrument, port, patch, and just about anything that's configurable on a micro level. Setting up tempo, punch in/out recording, loops, multiple channels, channel mutes, and solos are all straight-

forward. You can record in real time or in Step (pattern) mode, whichever suits your style. You can use clips (prerecorded snippets of MIDI and/or audio) by drag-and-drop or cut-and-paste, and you can link all of your clips so that any change made to one clip will happen to all of them.

Arranging and editing your tracks after recording is also intuitive and again the options are many. Markers can be added anywhere in the song, named accordingly, and accessed via key commands or toolbar icons with ease. MIDI effects such as arpeggiator, echo, filtering, and transposing are accessible via pop-up menus. Time Stretch and Shrink, which allow you to fit a section or sequence to a specified time or length, are great post tools for that one track that hangs a half-second past fade. Tell Pro Audio 9 you need to fit it within a certain time frame and it will do the rest. Audio tracks can also be changed by as much as 400 percent or as little as 25.

Quantizing is powerful and precise in Pro Audio 9. The resolution can be set to any note value and can be offset anywhere across a grid (adding or deleting, for example, five ticks in one measure) to shift the track in any direction.

You can even set parameters like Strength, Swing, and Window, which all effect which notes get quantized and by how much. Groove quantizing lets you apply an imported or constructed groove to any part of your song. Want to be on top of the beat, or a tick behind with every third kick drum? Set up a groove, apply it to the track, and start

grooving. There's also an option called Fit Improvisation that will create a tempo map without changing the feel or performance, for those times when you have a

killer idea and want to lay it down without setting up a tempo or time signature.

Drum sequencing and editing are greatly enhanced through the Session Drummer MIDI plug-in. Much like a regular drum machine, patterns and parts can be selected and edited with ease. Because it accepts standard MIDI files, different styles and patterns can be created and used in any song. Also new for Pro

Audio 9 is the Style Enhancer, another MIDI plug-in that adds feels to existing tracks. There are many styles and each is available for multiple instruments, which can greatly enhance a track with a feel you might not be familiar with.

Out of the Oven

Recording and editing was easy, and the program was stable as long as I didn't have any other programs running. I have a 400MHz Pentium II with 96MB of RAM, so bumping up to at least 128MB would help. There are a ton of extras, like the pile of MIDI songs covering every musical style imaginable. The plug-ins are also good, especially the Session Drummer and Style Enhancer, and can help inspire you when your creativity is running low. They are also great tools for seeing how a song is written, or how using controllers for pitch bend or other "feel" things are done. SYSEX control is well implemented, allowing you to load patches on all your gear every time you open a sequence. Once you learn how to set up your synths through SYSEX commands, you can save a lot of time by avoiding having to load all of your presets by hand.

With MIDI/audio integration, you can automate nearly everything. Stop and start anywhere in the track, and there's your mixer, built right in, with good quality effects and incredible control.



One possible working setup, showing the mass of toolbars across the top, the mixing console and piano roll view, and notation across the bottom.

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

Punching in and out of a guitar or vocal track is painless and automatable, and mistakes such as hand noise or hum can be edited and fixed in seconds. Pro Audio 9 has lots of interesting plug-ins, too, like the vintage amp sim, reverb, and the usual flangers and choruses. While it has its share of editing tools, there's support for an external editor (Sound Forge XP is included) to make professional editing a snap. There's even a way to turn audio information into MIDI note information on monophonic tracks, which can then be treated like any other MIDI track. The bottom line is, Pro Audio 9 integrates audio with MIDI quite well and includes some great tools to manipulate it with.

My main complaints are the window clutter and dull Windows interface. There's a lot of stuff going on and it's very busy, but they might have achieved the same level of control with a more streamlined interface. While the appearance is color-customizable, you're still stuck with a mundane interface. Looking

pretty doesn't get the job done any better, but it can definitely inspire greatness. How many times have you been turned on to a piece of gear because it looked cool or had some crazy colored lights? It would have been nice if the Console view's appearance had been reflected throughout the whole program. But in light of the program's functionality, the aesthetics are trivial: If it works, don't break it. And Pro Audio 9 works.

Other than these few minor UI complaints and technical-support shortcomings, Pro Audio 9 is everything it aims to be and certainly deserves its popularity. While there's not enough here to make me abandon my Mac, Pro Audio 9 would be my first choice for a top-notch PC sequencer. For the professional who needs to tailor his or her tracks with fine detail, this is a software package that can handle any project with great flexibility, strong performance, and the support of a number-one seller, so you can have your cake and eat it too. 🐉

PRO AUDIO 9 ★★★★★

STATS

CAKEWALK

Cambridge, Mass.

(888) CAKEWALK or (617) 441-7870

www.cakewalk.com

PRICE: \$429

SYSTEM REQUIREMENTS: 200MHz Pentium II, 64MB RAM for Windows 95/98; 300MHz Pentium II, 128MB RAM for Windows NT.

PROS

1. Easy setup for audio and MIDI with your PC's sound card.
2. Excellent collection of MIDI files, song styles, and tutorials; good plug-ins for MIDI and audio.
3. Extensive online support through newsgroups and users' groups.

CONS

1. Boring interface; extremely cluttered toolbars.
2. Lack of assistance in setting up external MIDI interfaces and complicated MIDI studios.
3. Poor e-mail support.

The Six Million Dollar Dolphin

Virtual Bionic Characters

As a kid who grew up on 1970s television, I was fascinated by the amazing advances in science shown to me every night. I watched the government establish a moon base on *Space 1999* and Leonard Nimoy telling me aliens may have built Stonehenge on *In Search Of...* I also knew that if you were important, good looking, and in a terrible accident, the government could rebuild you and make you better than you were, like in *The Six Million Dollar Man*. A lot of kids my age were fascinated with the idea of bionics. I am sure the boys at least pondered Lindsay Wagner and the implications of a bionic woman.

Here we are in the year 2000. Alas, 1999 came and went without a moon base. We still have no real idea how anyone could have built Stonehenge. However, it is starting to look like bionics may become a reality. The idea of stimulating nerve cells with electrical impulses has generated a lot of possibilities for treating physical disabilities. For example, cochlear implants have become a promising treatment for some types of deafness. More recently, Christopher Reeve's high-profile struggle with paralysis has opened many eyes to the possibilities of electronic stimulation of muscle tissue for locomotion.

Bionic Animation

When it comes to animation, we are still in the fairy tale days. Our characters are like Pinocchio. We control them by pulling on the limbs and bones directly, much as a puppeteer controls his marionette. Creating realistic-looking animation this way is very much an art. However, if we were to move a character by simulating the actual physical mechanisms that make a real person move, we would get realistic animation automatically. At least that's the theory. In practice, controlling a character through a physiological simulation would be a

major challenge. However, that doesn't mean we can't experiment with new techniques for locomotion.

Recently, I decided that I wanted to animate a dolphin. My typical approach would be to create a polygonal model and embed a simple skeletal system within the object which would be used to animate it. I could then create a set of animations to cause the dolphin to look like it was swimming and turning. However, this type of animation would not allow the dolphin to react to situations such as obstacles and changes in water flow. So I thought it would be interesting to attack the problem as a physical simulation.

Last month ("In This Corner... The Crusher!" June 1999), I used a mass-and-spring system to manipulate a free-form deformation lattice. Now, a dolphin is not exactly a soft-body object, since it has an internal rigid skeleton. However, dolphins

do move in a very fluid and soft manner. Demetri Terzopoulos has written about the use of a spring system to create artificial fishes (see For More Information section), so it seemed like a promising approach.

Building an Artificial Dolphin

I start with my dolphin mesh as you can see in Figure 1a. From this model, I created a low-level physical representation of the dolphin shown in Figure 1b. This control mesh represents the basic surfaces that I will need for the physical simulation. The key pieces are the tail fluke, which will provide the basic propulsion, as well as the dorsal and pectoral fins, which will keep the model from turning over while locomoting.

In order to create the physical simulation, I need to take this control mesh and



AUTHOR'S BIO | When not out hanging with the dolphins at the beach near his home, Jeff can be found at Darwin 3D thinking about the waves. Knock him off his board with a letter to jeffl@darwin3d.com.

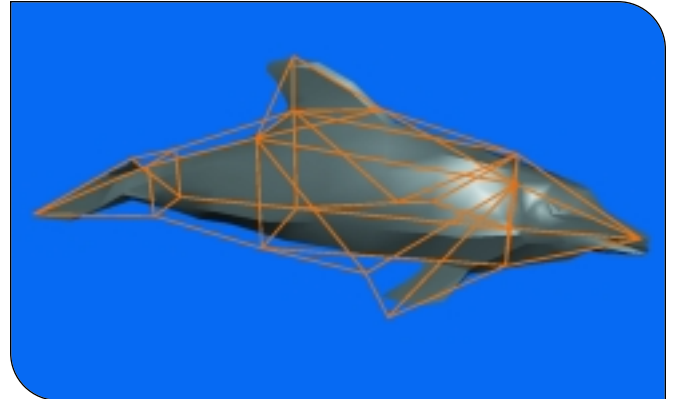
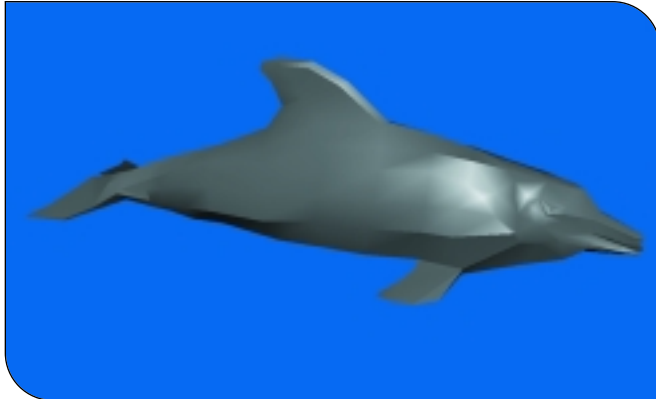


FIGURE 1A (left). The dolphin mesh we start out with.

FIGURE 1B (right). The control mesh we'll use in our simulation.

turn it into a mass-and-spring system. This is quite easy as I simply convert the vertices in the control mesh into mass points, and the edges that connect them become the base springs. In order for the object to maintain stability, I need to add crossbeam supports much as I did in the FFD simulation last month. Once this is accomplished, I end up with a mass-and-spring system that looks like Figure 2.

If I drop that into my physical simulation with the gravity turned off, it will

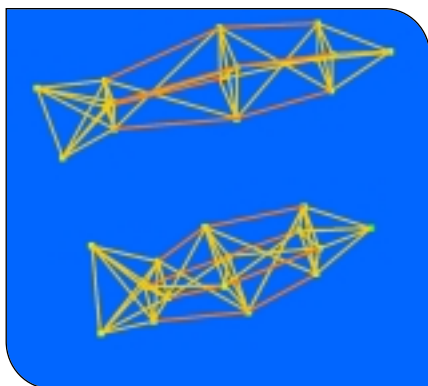
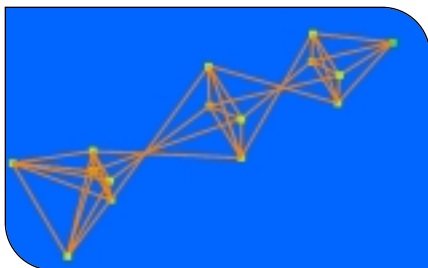


FIGURE 2 (top). The dolphin's mass-and-spring system.

FIGURE 3 (bottom). The muscles in action.

float through the simulation space. It will bounce off any objects or surfaces it collides with, generally acting like a blobby marine mammal that doesn't move much. In order to make the object move, I need to add some muscles.

Adding the Bionics

To get things moving, I want to create some virtual muscles. It seems that they should be positioned in place of some of the existing structural springs in the object, so let me take a minute to review the mathematics of these springs. The springs in the simulation are a combination of a linear spring force and a damping force as defined by Hook's spring law. This formula relates a spring force, f , between two particles, a and b .

$$f_a = - \left[k_s (|l| - r) + k_d \frac{\dot{l} - l}{|l|} \right] \frac{l}{|l|}$$

$$f_b = -f_a$$

$$l = a - b$$

$$\dot{l} = v_a - v_b$$

The stiffness of the spring is defined by the spring and damping coefficients, k_s and k_d . The value l is known as the rest length of the spring. This defines the desired distance between the two particles as set in the initial mesh. This rest length is the key to creating a virtual muscle. If I were to dynamically decrease the rest length of a spring, it would begin to contract, causing a force that brings the two particles together.

I can define a maximum contraction amount for each muscle. For example, I may want a muscle to contract to 50 percent of its rest length. The maximum contraction becomes 0.5 times the rest length. I can then use a percentage in the range of 0 to 1 as an activation value to make the muscle go from rest to the 50 percent length target.

Figure 3 shows the control mesh with some springs activated as muscles, marked in dark orange. The first frame shows the start of the simulation, then the muscles start contracting to create the second frame.

Controlling the firing of the muscles is a job for some form of primitive AI system. A dolphin swims by contracting muscles that cause the animal to arch its back in an alternating up-and-down motion. I can simulate this by first contracting the muscles along the top and then the muscles along the bottom. This is most easily accomplished with a sine wave that is scaled into the range of 0 to 1. I then offset the phase of the two sets of muscles by 180 degrees. That way, while the top muscles are contracting, the bottom muscles are relaxing. I can control the speed of contraction by increasing the wave frequency.

Moving Through the Water

The use of this primitive brain to cause muscles in the simulated dolphin to contract achieves the effect of making the dolphin bend back and forth. However, it does not really move anywhere in the simulated environment. That's because there

are no forces actually causing any propulsion. The key to making the dolphin swim through the water is hydrodynamics. I need the movement of the dolphin's tail fluke to displace virtual water and cause forward motion.

You can see how this happens in Figure 4. Vector n is the normal to the surface of the control mesh. Since the positions and velocities for all nodes in the control are calculated in the simulation, I can examine directly the velocity vectors of each node in the system, for example v_a and v_b . For each triangle in the control mesh, let me use the formula

$$f = \min\left[0, -\frac{A(n \cdot v)n}{3}\right]$$

where A is the area of the control triangle being tested. As you can see, if the velocity of a particle in the control mesh is moving in the direction of the surface normal, I will get the maximum force on that node. The force is divided by three, since three nodes make up a control triangle. If the velocity vector is either pointing in the opposite direction from or tangential to the surface, no displacement force is created. This is exactly the behavior that I want.

However, I need to determine the surface area of the control mesh triangles quickly. Since the object is a soft body, the positions of the control mesh vertices change during the simulation. I will also need to calculate the surface normals for each triangle.

A very common 3D graphics operation is calculating the normal to a triangle by using the cross product of the two vectors that make up the triangle. However, another key benefit of the cross

product is that the magnitude of the cross product of two vectors is equal to the area of the parallelogram that the vectors describe, as you can see in Figure 5.

So the area of the triangle is equal to half the magnitude of the cross product. That means I can rapidly calculate the normal of a triangle as well as get the area of the triangle by using the cross product.

I can use this information to calculate the normals and the area at each control triangle and apply the hydrodynamic force to each control vertex. The dolphin now swims forward and the aerodynamic design of the control mesh automatically serves to keep the dolphin upright. In fact, I can even use this aerodynamic capability to control the direction the dolphin swims.

Controlling the Animal

Dolphins move by subtly changing the direction their tail is moving as well as changing the direction of their head and the attitude of their pectoral fins. I can control my virtual dolphin the same

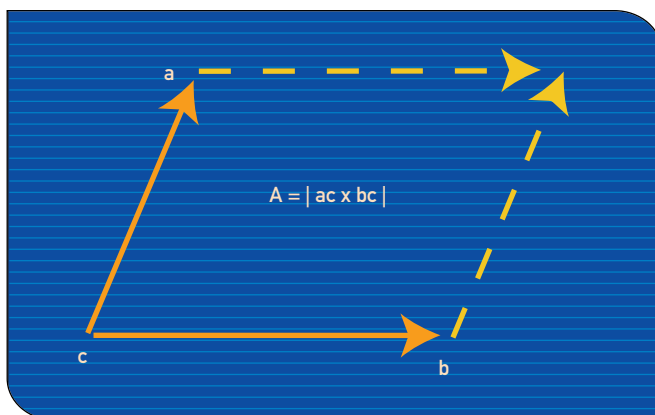
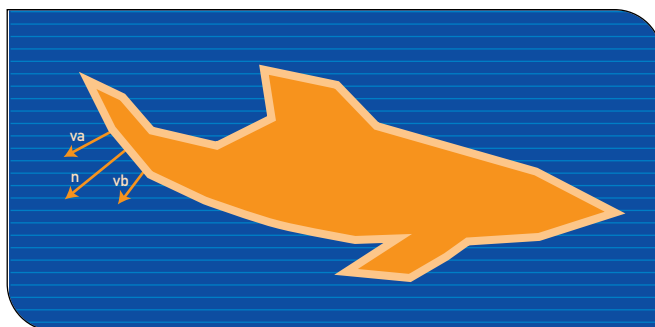


FIGURE 4 (top). The dolphin in motion.

FIGURE 5 (bottom). Area of a parallelogram.

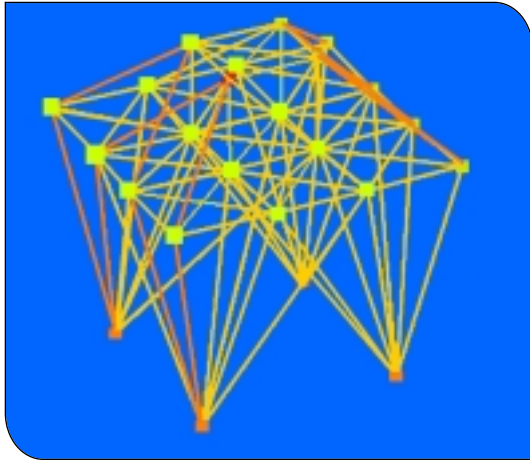


FIGURE 6. The walker. This creature relies on friction to propel itself.

way. Direction-control muscles can be activated on the neck and pectoral fins to guide the steering of the dolphin much in the way that one might maneuver a radio-controlled glider. These same controls can also be used to adjust the pitch of the pectoral fins, causing the dolphin to swim lower or higher in the water column. I have found that these muscles need only contract very slightly to affect the animal's direction.

In his paper, Demetri Terzopoulos describes using a learning system to optimize the motor-control parameters to achieve goals like maximum speed efficiency. I did not experiment with any learning algorithms for the dolphin, choosing instead to tune the parameters by hand. I think a learning system would greatly improve the performance and it's something I would like to investigate further in the future.

These low-level steering controls could be combined into high-level behaviors to achieve things such as schooling and threat-avoidance among multiple synthetic dolphins. It's clear that many of these techniques can be applied directly to real-time game simulations.

Final Display

Once the dolphin is moving with the muscle system, I want to render it. I could just render the control mesh directly, though it is not very interesting visually. However, the control mesh can be used

to determine the position of the bones in the animal. I position bones at key locations along the animal's spine and use the position of the control mesh to determine the orientation of the bones. These bones are weighted to the display mesh based on proximity and the mesh is rendered using the same matrix deformation system that I used in the FFD demonstration last month. This gives me a visually detailed render model that is animated using the simple control mesh.

Other Creatures

I also spent some time designing other creatures. Instead of a swimming animal, I created a four-legged walker, which you can see in Figure 6. This creature makes use of a friction model to move itself forward. However, the control sequence for the muscles on this creature is much more complicated than for the dolphin. I had a very hard time producing any gaits besides a simple walk. This is a case that would benefit greatly from an automatic learning system for the animation sequencer.

It's relatively easy to create a creature in a modeling package and bring it into the simulation. Attach a few springs and muscles, and away it goes — usually collapsing into a heap of vertices on the floor. However, achieving a stable object is pretty easy. The tricky bit is coming up with a locomotion algorithm that works as well in practice as it did when you thought it up. I would like to try a slithering gait based on directional friction as Gavin Miller described in his research on snakes and worms (see For Further Info). It seems like that would look very interesting in a real-time game and I'm confident the performance would be much better than the 30 seconds per frame that Miller needed when he did his work in 1988.

Other Methods

Rather than using the soft-body muscle-based technique for animating the characters, I could have driven the skeleton of the dolphin directly. In place of

using springs as muscles, I could have used torques to dynamically change the orientation of each joint in the character. That would have probably simulated how an actual dolphin moves more accurately, but that would have required me to use a fairly complex articulated body solver to determine the dynamics of the system. If you are interested in pursuing that direction for dynamic animation, I suggest you check out Jessica Hodgins' and Chris Hecker's presentations from this year's Game Developers Conference. That material will get you going with a hopper at least, if not an articulated dolphin.

For now, play around with the muscle simulator and see what you can create. This application can be greatly improved by creating a more robust controller UI. The control parameters could also be optimized through the use of a learning technique as I mentioned above. Get started by grabbing the source and application off the *Game Developer* web site, www.gdmag.com.

FOR MORE INFORMATION

- Terzopoulos, Demetri, Xiaoyuan Tu, and Radek Grzeszczuk. "Artificial Fishes: Autonomous Locomotion, Perception, Behavior, and Learning in a Simulated Physical World." *Artificial Life* Vol. 1, No. 4 (December 1994): pp. 327–351.
- Miller, Gavin S. P. "The Motion Dynamics of Snakes and Worms," *Computer Graphics* Vol. 22, No. 4 (August 1988): pp. 169–173.
- Hecker, Chris. "Simulating a Locomoting Character." Game Developers Conference 2000. Go to www.d6.com/users/checker/index.html for more information.
- Hodgins, Jessica. "Controlling a Locomoting Character." Game Developer's Conference 2000. Go to www.gvu.gatech.edu/~jessica.hodgins/ for more information.

SODAConstructor

www.sodaplay.com/constructor/index.htm
This is a cool application on the web that quite a few of you have written me about. It allows you to create a simple 2D mass-and-spring object and animate it using muscles in a nice Java interface.

Beauty and the Beast

Morphing Mayhem

Morphing between 3D objects is not all that new to computer graphics. If you have seen even one big-budget, blockbuster Hollywood movie in the last five years you have seen 3D morphing in action. However, the technique holds exciting new possibilities for those of us working with real-time 3D graphics. As gaming technology closes the gap between prerendered movies and real-time 3D, we will see morphing used for special effects and magical transformations more often. As artists continue to experiment more with morphing, new problems and techniques are sure to come up. This article will explore one such problem, morphing between objects that have different numbers of vertices, and the techniques that came out of working around it.

In games today, the most prevalent use of morphing is for bringing characters to life through facial animation and lip-synch. This is done by creating morph targets of the character smiling, frowning, mouthing different phonemes, and so on, and morphing between them. This is a relatively easy process for an artist to set up, at least from the point of view of getting the morphing to happen. However, animating it and making it believable is a whole other art form, and the subject of another article. In its most basic form, the procedure for setting up a character model for morphing lip-synch is as follows. First, make as many clones of the

object as there are morph targets. Next, carefully adjust the key vertices on each new model to form different phonemes and expressions. Finally, animate the model, morphing between the targets. Animation is usually set up with the help of a morphing utility that allows you to assign each morph target to a separate channel and animate a blend between several targets. 3D Studio Max 3 ships with a modifier called Morpher that works in this way.

The above example, with the same number of vertices in an identical order, is the simplest scenario for creating a morph. By using a clone of your object for each morph target you are ensuring that the morph will work, but what if you want to use morphing for something other than lip-synch? What if you need to morph between two or more separate objects that were created independently of each other and have different vertex and polygon counts?

Enter the party girl and the beast (Figure 1). Is it possible to take a 1,770-polygon, 921-vertex model of the party girl and morph it into a 1,068-polygon, 536-vertex gorilla head? I explored several possible ways to accomplish this with 3D Studio Max (other 3D packages will most likely have similar tools). But before we get ahead of ourselves, let's start with some basics about morphing.



FIGURE 1. Morphing a woman's head into a gorilla's presents a formidable challenge because of their different vertex and polygon counts.

Morphing Basics

In simplest terms, morphing is the process of taking one shape and transforming it into a different one. In the world of 3D object creation, there are two basic requirements for setting up polygon-based models for morphing. The first is that all the objects that are going to be used for the morph must have the same number of vertices in each piece of geometry. The second is that the vertices in each model must be arranged in the same order, which I will refer to as surface order.

Why is this? Behind the magic of morphing is a software algorithm that takes the physical features of one object (Object A) and gradually transforms them into the features of another object (Object B). Obviously, a computer doesn't understand which of Object A's features correspond to which of Object B's, so we have to specify that explicitly. Giving an object's vertices a specific number that has a corresponding number in the object that it is morphing into accomplishes this. The computer then knows to move Object A's vertex 20 to vertex 20 on Object B. If A's vertex 20 is in the upper-right corner of the model and B's vertex 20 is in the lower-left corner, then Object A will appear to flip upside down as it changes into Object B. If there are more vertices in Object A than in Object B, you will not be given the option of morphing at all. This is why the lip-synch example above works so well. By taking clones of the original model and simply moving vertices around, being careful not to add or delete any, you are ensuring that the models will all have the same vertex count and surface order. Your character will spring to life and the morphing will work right away.

This brings us to our current dilemma of morphing between the human and the

AUTHOR'S BIO | Lisa Washburn has successfully morphed back into her real job, running her RT3D art production company Vector Graphics (www.vector.org.com). Send comments and questions to article@vector.org.com.

gorilla head. As the models stand now, we do not have the option of morphing them. Before we begin looking at how we can make these two objects morph, let's take a closer look at the models.

The gorilla head model — let's call him Bobo — consists of four separate pieces of geometry. There is a separate sphere for each eye, a piece for the lower jaw, and then the bulk of the head. For the sake of simplicity, I'm only going to morph the bulk of the head, which as I mentioned before has 536 vertices and 1,068 faces. The human female head, which we will name Vivian, is all one piece of geometry with 921 vertices and 1,770 faces.

So what are our options? To be able to morph, we must first rebuild the models so that they have the same vertex count. If you are starting as I am with models that have already been textured and mapped, this isn't the best news, as you will have to re-create the mapping. If you are using Max and you haven't collapsed the object's modifier stack, simply make a clone of the original model, hide it, and use it later to acquire the mapping information. There are four different procedures that we'll try in order to rebuild the models so they share the same number of vertices in the same order.

Cloning

The first and most obvious procedure is to clone one of the objects and rearrange the vertices until they match the other object's. As long as you don't divide any edges or weld up any vertices, you are guaranteed a model that will morph. This works very well for objects that are similar in shape, size and vertex count, like the lip-synch example mentioned above. However, for radically different models, particularly those with very high polygon counts, this can be an extremely tedious, time-consuming, and messy operation. To apply this to my example I cloned Vivian, who has the higher vertex count. After scaling the model up to be a bit bigger than Bobo's head, I began using Max's 3D Snap tool to snap vertices into place. After I got four or five into place, I began to realize that I would have to do this 921 times! Then, I would have to turn edges and deal with where to put the extra vertices. Even if I used Bobo's head as the clone, I would still have to

move 536 vertices by hand. This is clearly not an elegant solution. What if I had 100 models that I needed to morph? What if the models were 30,000 vertices each? There must be an easier way.

Shrink Wrap

The next option came out of a brainstorming session with some colleagues on possible ways of automating the cloning option above. How could we maintain the vertex count and surface order that results from cloning the object, yet not have to go

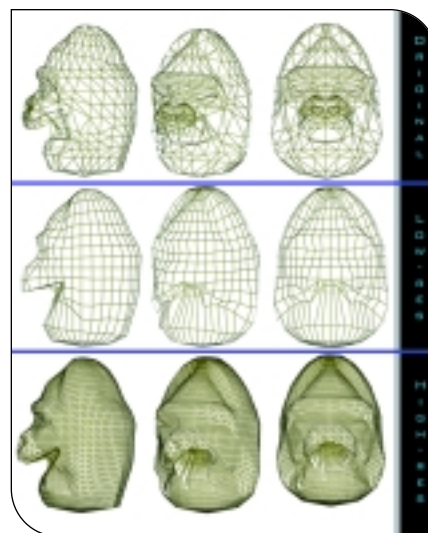
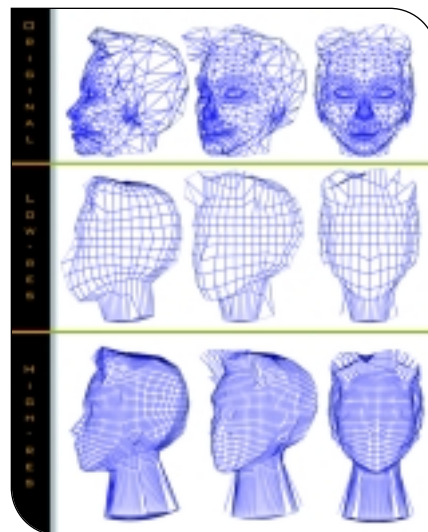


FIGURE 2. Sphere conform results for Vivian (top) and Bobo (bottom), showing the original model, the low-resolution results, and the high-resolution results.

through the tedious effort of moving each vertex manually? Shrink-wrapping popped into the conversation. What if you had an object, for example a sphere, that you could shrink wrap onto other objects? If you took clones of the same sphere and wrapped them around different objects, theoretically you would be able to make perfect copies of your original objects with the same vertex count, and therefore be able to morph them. Several 3D packages already have a similar tool. In Max it's called Conform and works by projecting the vertices of one object onto the surface of another. You are given a couple of choices on what direction you want the vertices to be projected. Max also has a Conform Space Warp, but the Conform compound object is more applicable to what we are trying to do.

There are two different ways to approach a procedure like the Conform tool. One is to create an object independent of the models that you are interested in morphing, making a clone for each model, and then conforming each independent object to a different original model. The main advantage to this technique is that you can control the number of vertices in the final object. If neither of your original models has the vertex and polygon count that you need for the final objects, then this technique can get you the numbers that you need. To test this out on my two heads, I created a sphere of 482 vertices and 960 faces and then cloned it. I then selected Bobo's head, aligned and centered the pivot point, and did the same to Vivian's head. I then aligned a sphere to each of the original objects and scaled them to be a bit larger than the models that they would be conforming to. I generated the conform objects using the Along Vertex Normals option and was struck by two things. The first is that the Conform tool has a bug that occasionally combines the original model into the conform object. The other is that the resulting model was blobby and missing the details that I had so carefully modeled into my faces, as you can see in Figure 2. I was able to get around the bug by clicking the Hide Wrap-To Object under Update and setting the Standoff Distance under the Wrapper Parameters to 50 and then back to 1. The blobbiness problem was not so easy to get around, however. Theorizing that the problem might have been caused by the sphere

having fewer vertices than the object that it was conforming to, I replaced the 482-vertex sphere with a 3,282-vertex one. This worked well for Bobo's head with the Conform object, even getting the modeled nostrils. The result for Vivian's head, though, wasn't much of an improvement over the lower-resolution sphere. The detail in her mouth and nose are completely gone. Another issue to take into account is that Vivian had originally been modeled with vertices placed in key facial muscle locations for lip synch and facial animation. This setup is completely missing from the uniform mess of the Conform compound object. Ultimately this technique did morph, and might be useful for objects that don't have a lot of detail. Still, the resulting models were not close enough to the originals for my purposes.

Another way to use the Conform tool is to make a clone of one of the original models and wrap it onto the other model. I tried this first by making a clone of Bobo's head, aligning it to cover Vivian's head completely, and clicking the conform button. The resulting model was very strange, as you can see in Figure 3. Some of the details of

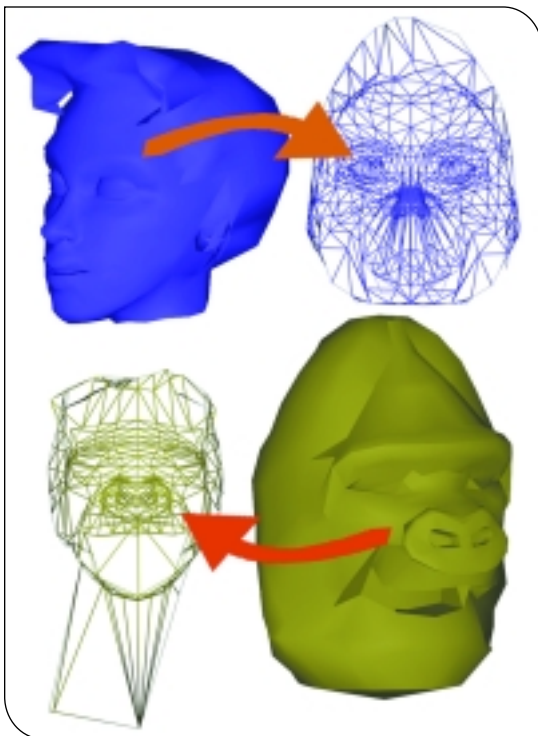


FIGURE 3. Conforming a clone. This technique produced some odd-looking results, but still has promise for the future.

the original model, such as the vertices that make up the nostrils of Bobo's nose, were still preserved in the new model, simply having been flattened out and pushed into the shape of Vivian's nose. Working in the opposite direction, Vivian into Bobo, produced the same strange results. Although the conform or shrink-wrap concept has potential, its execution needs to be more specific. There needs to be a more intelligent way of figuring out where the vertices should go when they wrap to the second object.

Cross Section

Another rebuilding option that came out of our brainstorming session was the use of cross sections. What if we had a black box that would take each model, slice it into a specific number of cross sections, normalize the number and order of the vertices on each cross section, and regenerate the model? This would allow us to create a number of models that have a set number of vertices in a set order. I set about figuring out how to do this in Max.

Cutting the model into cross sections was a piece of cake with Max's Section Object. Basically the Section Object is a rectangular plane that you move and rotate to where you want to generate a cross-sectional spline. You then hit the Create Shape button under the Modify panel and you get a 2D outline of your model at the point through which the Section Object was slicing. This is illustrated in Figure 4. The next question is how to take the cross section shapes and normalize the vertex count and order. This took a lot of searching through Max plug-in sites on the web, but I finally found a plug-in called Hspline from Habware (www.habware.at/duck3.htm). This shareware plug-in allows you to normalize the segments of the spline either by the number of segments or by segment length. By selecting all the splines and applying Hspline, you generate a set of shapes that all have the same number of vertices and all the vertices will be evenly

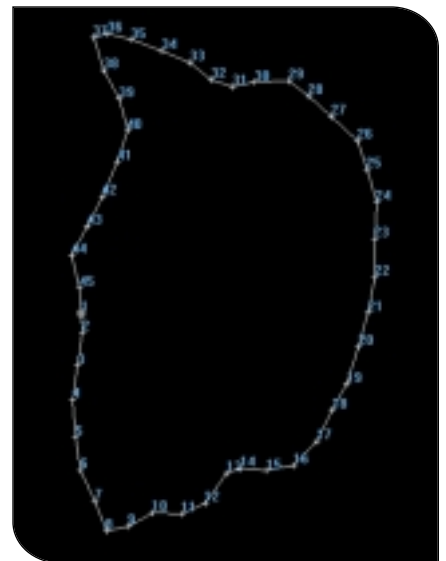
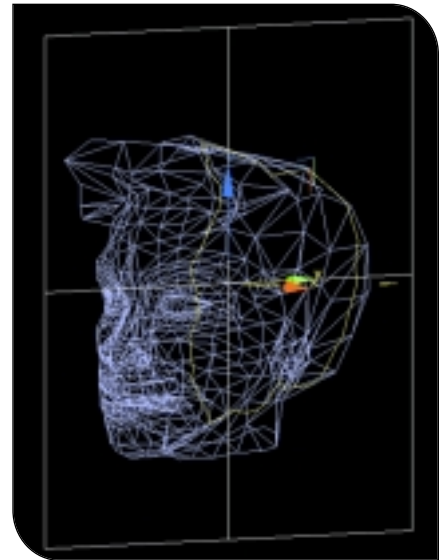


FIGURE 4 (top). The Section Object.

FIGURE 5 (bottom). Normalized spline with vertex numbers.

spaced apart, which you can see in Figure 5. By creating the same number of Section Objects for each model that you want to morph, and with each Section Object having the same number of vertices, you will be creating models that will be able to morph.

The next issue is generating a model from the new splines. Searching through Max again I came up with the Cross Section modifier. This modifier generates a spline skeleton or cage by connecting the vertices of different shapes in the same object. It is also dependent on vertex number and order,

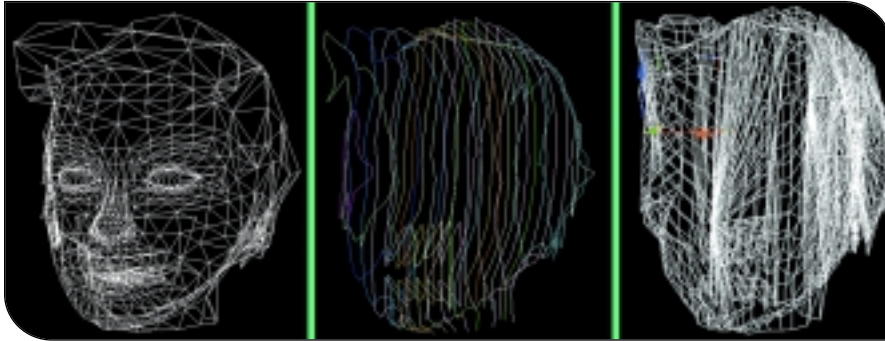


FIGURE 6. Left: Original model modeled in wireframe. Center: Cross-sectional splines of the head. Right: Initial results of Cross Section modifier.

so the Hspline plug-in was a real lifesaver for this part of the procedure as well. Having the same number of vertices on each spline makes the Cross Section modifier work much more smoothly.

One tip for using the Hspline to facilitate the Cross Section modifier is to go through each spline and check for three things. First, verify that there is indeed the number of vertices on each spline that you dialed in, as sometimes Hspline puts an extra vertex on top of the first vertex. Second, make sure that the order of the vertices goes in the same direction for each spline. If the order of vertices is clockwise on one spline and counterclockwise on the next, the Cross Section modifier will create a twisted spline cage. To access the vertex numbers in Max, select the spline, go to Sub-Object > Display, and click the box that says Show Vertex Numbers. To reverse the order of the vertices, go to Sub-Object > Spline and hit the Reverse button. The third thing to look for is whether the number-one vertex is in approximately the same location for each spline. Cross Section uses the vertex numbers to connect vertices, so if the first vertex on each spline is in a different place you'll again end up with a twisted spline cage.

Before you apply the Cross Section modifier, you have to attach the splines within each object. This is very important, as the order in which you attach the splines to each other determines the surface order of the vertices in the object. For example, if you attach the splines from bottom to top in Object A, and top to bottom in the Object B, when you go to morph A into B the model will turn upside down. If you attach both models top to bottom, then the model will remain right side up. A quick

warning here is that if you attach the splines in a haphazard order, for example you attach the first to the third, then to the second, then to the fifth, and so on, the Cross Section modifier will generate a cage that follows that order, generally resulting in a twisted mess.

After you apply the Cross Section modifier, which gives you the spline skeleton, the next step is to apply a Surface modifier. The Surface modifier generates a patch skin based on the vertices in the spline skeleton. Taken together, the Cross Section and Surface modifiers are referred to as Surface Tools in Max. The nice thing about the surface modifier is that it generates a patch object that gives you the option of dialing up or down the steps in the Patch Topology. This increases or decreases the complexity and ultimately the face count of the model.

I tried this procedure with two simple shapes and it worked beautifully, so the concept is correct. Does it work for models as detailed as Vivian and Bobo? In a word, no. Basically, I had the same problem with the rebuilt model with this procedure as I had with the Conform tool — the resulting model is not close enough to the original, as you can see in Figure 6. Also, this procedure requires a lot of tweaking and manipulation of the splines to get the Cross Section tool to create a spline cage that is not severely twisted and therefore unusable. This brings us to the fourth option.

Plug-ins

Surprisingly, I found no commercially available Max plug-ins to accomplish this task. I did find one shareware plug-in, but it was very difficult to set up and did-

n't work in the end. I posted to several mailing lists asking if anyone knew of any and didn't get a single response. Obviously there is a need waiting to be filled (hint to those of you more industrious readers out there). If I were writing a wish list for this plug-in, I would want something that captures, as close as possible, the original location and placement of the vertices on the original model, as well as allows morphing of mapping coordinates. A tall order perhaps, but it never hurts to ask.

Where We Stand

So can a wild-eyed party girl from the wrong side of the tracks mutate into a snarling beast? In a nutshell, not quite yet. Morphing between two independently created, highly detailed character models with different vertex counts is not an easy task. It's possible if your models are simple shapes and you don't have a lot of specific detail that you need to preserve. Many models that you might want to morph do fall into this category, for example a sack of gold into a sword, or one type of car into another. Even morphing Bobo's head into a less detailed character would have worked with the shrink wrap technique.


However, rebuilding models that are highly detailed and need to have vertices in specific places (for example if you need to create the effect of facial muscles for lip-synch, like with Vivian) is still difficult to automate. We need a tool that automates and streamlines this procedure. Imagine the story line possibilities of being able to morph easily between characters in a game, characters and other objects (such as cars, dinosaurs, or dust clouds), or into a character from a different game altogether. Think about the production time you would save if you could take a commercially modeled object and morph it with one you created. The possibilities are endless, and hopefully just around the corner. 🐾

ACKNOWLEDGEMENTS

Special thanks to Michael Hultner of Anatomix (www.anatomix.com) and Kent Suzuki of Right Brain Electronics (www.rightbrainelectronics.com) for helpful discussions during the preparation of this article.

Go with the Flow:

AUTHOR'S BIO | *When he's not sitting around radiating potential, Brian's probably busy furthering the secret OpenGL agenda. Either that, or he's likely doing the same thing he does every night, Pinky — trying to take over the world. Send preemptive bribes and/or tribute to brian@maniacal.org.*



Improving Fluid Rendering Using Implicit Surfaces

Maybe it's just me, but it seems like games have always had something of a love-hate relationship with fluid. We've been using water, lava, and the like for a very long time. For many years, games such as *ULTIMA UNDERWORLD* had 3D-rendered water and lava — rendered as flat, textured planes. At the time, that was an impressive part of a very impressive engine. Today, that engine is objectively obsolete, having laid the initial groundwork for today's flashiest polygon-pushers.

Illustration by Spencer Lindsay.
Created with RealFlow In 3D Studio Max

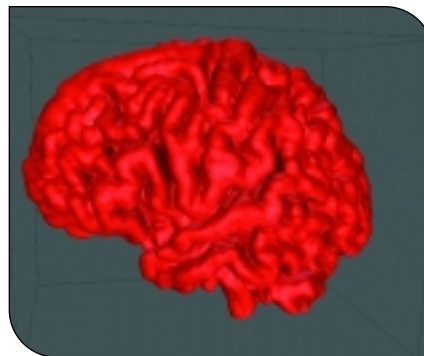
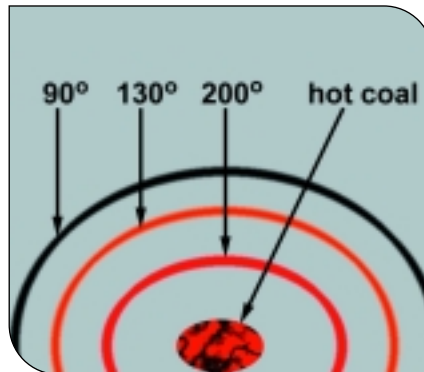
In spite of other advances in game technology, however, the flat-plane approach to water has remained fairly unchanged. Perhaps today the water plane is drawn transparently using an innovative blend mode. Perhaps it's broken into more polygons and jiggled up and down a bit. But the underlying technology (or lack thereof) has been the same since the advent of real-time 3D game engines.

It's a bit of a pet peeve of mine, because more advanced fluid rendering opens up the door to some very cool game-play dynamics. How great would it be if in a 3D real-time strategy game you could flood an enemy base by diverting a river through their valley? Or if your opponent's base were hydroelectrically powered and you built a dam upstream to cut off the river flow, shutting down his power? These are the kinds of advanced interactions that a jiggly plane just can't offer.

So this month and next, I'll take a stab at describing one technique for advanced fluid rendering for games. Note that I'm not the only person who wants better water: Jeff Lander has written a few columns on fluid flow, including overviews of the Navier-Stokes equation and various simplifications thereof ("A Clean Start: Washing Away the Millennium," *Graphic Content*, December 1999, and "Research on the Rhine: Reflections on Water Simulation," *Graphic Content*, January 2000). Rather than build off of that work, though, I'm using a different technique, a surface representation known as implicit surfaces.

Implicit What?

An implicit surface can be succinctly described as an isosurface of a real-valued 3D function. If that doesn't make it crystal clear, perhaps an example will help. Imagine a hot coal sitting on the ground, radiating heat outwards. In this case, temperature is our real-valued 3D function: at every point in a 3D space (the room the coal is in), the temperature is a real number — what your programming language of choice might call a float.



If you pick a specific temperature value — say, 90 degrees Fahrenheit — and paint every molecule at that temperature bright red, you'll end up with a surface that looks something like a spherical shell around the coal. The higher the temperature you pick, the closer that surface will be to the coal. That's an implicit surface, and the temperature you choose is the isovalue. Figure 1 shows a number of the isosurfaces in 2D.

The name "implicit surface" is then fairly straightforward. It's a surface because it's a surface, of course, and it's implicit because we don't have an explicit parameterization for it; the 3D function and the isovalue imply where the surface is.

And These Are Good For What?

Granted, it's not immediately obvious what you'd use these isosurfaces for. After all, how many useful real-valued 3D functions are just sitting around waiting for you to slap an isovalue on them? It's worthwhile, then, to take a look at what isosurfaces are used for in graphics.

One use of implicit surface rendering is

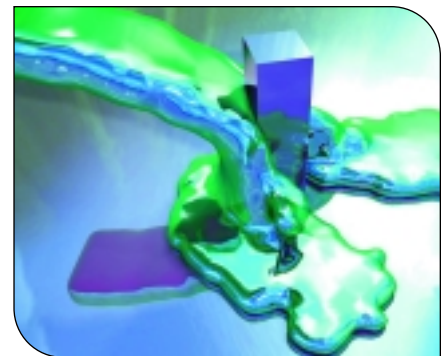
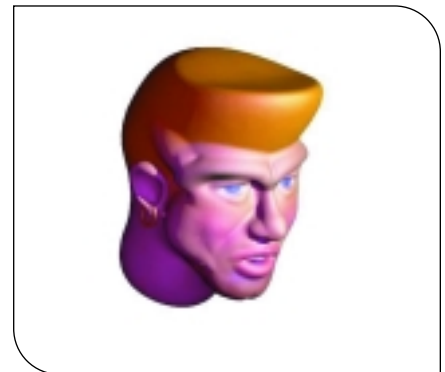


FIGURE 1 (top left). A hot coal radiates heat. Three different isosurfaces are labeled by temperature. **FIGURE 2** (bottom left). A human cortex rendered using implicit surfaces (image courtesy of Paul Bourke). **FIGURE 3** (top right). A character modeled using metaballs/blobby modeling in *Organica* (image courtesy of *Impulse Inc.*). **FIGURE 4** (bottom right). Fluid flow rendered using implicit surfaces by *RealFlow* (image courtesy of *Next Limit / RealFlow*).

in medical visualization. Data from an MRI scan, for example, provides a sampled 3D function of intensity values over some tissue. The various intensity values highlight types of tissues. For example, one might best highlight bone, whereas another might depict cartilage particularly well. Clearly, I'm glossing over quite a bit here. Figure 2 shows a human cortex rendered with implicit surfaces from MRI data. For more information on that specific application, see the For More Information section at the end of this article for Paul Bourke's web site address. Another use of implicit surfaces is often referred to as "blobby modeling." This is the form of implicit surface modeling that's common to many 3D

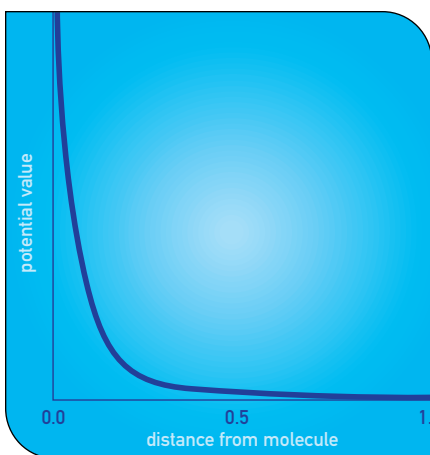
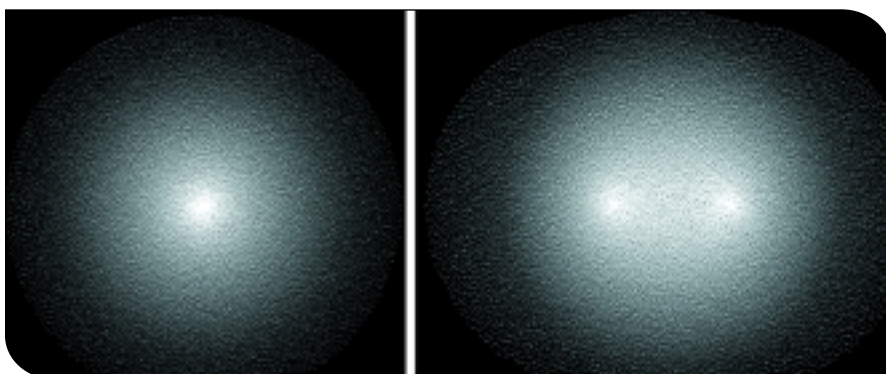


FIGURE 5A (top left). A single molecule radiates potential outwards. **FIGURE 5B (top right).** Two molecules radiate potential: the potential fields sum where they overlap. **FIGURE 6 (above).** The potential function, $(1/d^2) - 1$. Past a distance of 1.0 the value is clamped to 0.

modeling packages, and it also goes by monikers such as “metaballs” or “blobs.” The basic idea is to define the implicit function with primitives such as points, lines, boxes, and other geometric shapes. These primitives radiate potential similar to the coal radiating heat. That way, when you place two primitives near each other, their potential fields sum and the surface flows smoothly between them. This outcome tends to produce organic or cartoony results, and indeed it is good for modeling things such as organic or balloon characters and clay figures. Figure 3 shows an example of a character modeled in Organica (see the For More Information box to find details on Organica).

The final use of implicit surfaces is fluid

modeling, which is, not coincidentally, the focus of this article. The idea is to model fluid flow through molecular interaction. Using a reduced number of molecules, you treat each molecule as a point primitive in the implicit function, just as you would point primitives in blobby modeling. That way, the molecules generate an implicit function. If you pick an isovalue that produces a surface of the desired volume and move the points around using some physical model of molecular interaction, then *voilà!* The technique also scales quite well — turning up the number of molecules and tightening the isovalue in on the surface will produce a higher-quality approximation of the surface. There’s an excellent non-real-time tool available called Real-Flow, shown in Figure 4. Clearly, since this fluid rendering is non-real-time, the number of molecules can be cranked up quite a bit.

Hopefully that’s enough to convince you that implicit surfaces are useful enough to merit a closer look. Now that you have a general overview, let’s dig into the specifics of our implicit surface model.

The Implicit Function

Details, details... I’ve mentioned how all these points “radiate potential” and how the potential “sums” to produce smooth surfaces, but I haven’t described exactly how it’s going to happen. So, I’ll describe the process step by step to try to keep everything straight.

To start with, one thing that may cause confusion is the concept of “potential”

radiating from these molecules. Just what is potential? Well, it’s really a unitless value and doesn’t represent a meaningful quantity such as pounds or liters, so perhaps an analogy might help. Think of the molecules as radiating some matter — say, dust — outwards. The dust is very concentrated near the molecule, but as we move away from the molecule, the dust thins out and is much less dense. In this scenario, the potential at a certain distance from the molecule is just the density of the dust. Figure 5a shows a 2D version of this.

From there, it makes sense that when we put two molecules near each other, their density fields add up, producing a thicker dust cloud between them, as illustrated in Figure 5b.

However, I haven’t specified at exactly what speed the dust thins as it moves away from the point. How to decide? Well, we can use any function we want; it’s just a question of whether it fits our needs. Our only real need is for it to look good, which entails a number of things. We want the fall-off to be at least C0 continuous, of course — that is, it shouldn’t have any abrupt changes in value — or else the surface might not be smooth or it might even have open holes.

Since liquids are noncompressible, the ideal fall-off function would preserve volume. This means that if we have two molecules, the surface produced when they’re far away from each other should have the same volume as the surface produced when they’re near each other. Unfortunately, this is really hard to do in practice. In fact, there’s no single fall-off function that will do this for us. Therefore, you should pick a function that keeps the surface volume from changing too abruptly.

There’s actually a fair amount of research that’s been done in this area and a number of suggested fall-off functions from various authors, but I’ve chosen to eschew the well-traveled path and just make up a function that looks O.K. Given some distance d away from the molecule, the value is equal to:

$$\text{potential}(d) = \left(\max\left(0, \frac{1}{d^2} - 1\right) \right)$$

A graph of this function is shown in Figure 6. This function has a number of desirable characteristics. The first is that while it does require a division, it uses the distance squared, which you can get from a vector without a square root. (You'll be evaluating this function so many times at run time that these kind of low-level details really are significant.)

Another benefit of this function is that the potential clamps to 0 at a fixed distance of 1.0 from the molecule. This means that the molecule only influences the area within a distance of 1.0 and after that it doesn't have any effect. In next month's article, I'll explain exactly how handy this is, as it allows us to partition the molecules spatially, greatly simplifying the job of evaluating the implicit function.

To generalize the function for a single molecule to a multi-molecule surface, we have to change the arguments a little. Let p be the point in space for which we want to evaluate the implicit function. Then let c_i be the location of the i th molecule (where the total number of molecules is n). Then, the value of the function at p is:

$$\text{potential}(p) = \sum_{i=0}^n \max\left(0, \frac{1}{|c_i - p|^2} - 1\right)$$

In case you read code more easily than math, the equivalent pseudocode for that function is shown in Listing 1.

That's it for the implicit function. The only remaining part of the implicit surface representation I haven't discussed yet is the isovalue, and for good reason: there's not really much to it. Just pick a value, and if the resulting surface doesn't look good enough, pick some other value. Higher isovalues will tighten the surface in on the molecules, making the surface seem to have a lower surface tension. Lower isovalues will expand the surface away from the molecules and make it look like it has a higher surface tension. Of course, the higher the isovalue, the more molecules it takes to produce a surface of a given volume, so the trick is in finding a good compromise somewhere in between.

LISTING 1. Pseudocode to calculate the potential value at a point given a number of molecules.

```
potential(Point p)
{
    float totalValue = 0

    foreach molecule
    {
        float distance = distanceFrom(molecule center, p)
        float contribution = (1 / distance^2) - 1
        if contribution > 0
            totalValue += contribution
    }

    return totalValue
}
```

Rendering

So far, I've covered enough information that if someone handed you a bunch of molecules, you could evaluate the implicit function defined by those molecules at various points in space. That's wonderful, but it doesn't help much with the real task at hand, which is rendering isosurfaces of that implicit function.

There are ways to raytrace implicit surfaces directly; packages such as POV-Ray (www.povray.org) yield pixel-perfect renderings of the surfaces. Unfortunately for us, though, consumer-level 3D hardware isn't quite up to the level of real-time raytracing, or even automatically scan-converting implicit surfaces. Therefore, we need a way to convert our implicit surface into primitives that the hardware can understand, namely triangles.

There are a few different algorithms for doing this. One starts with a rough shell

that contains the surface and subdivides the shell until it conforms to the surface within some particular error threshold. My experience with subdivision surfaces suggests that the subdivision process would be prone to lots of memory-thrashing and difficult to optimize. Furthermore, maintaining all the connectivity information to subdivide a mesh is a lot of confusing, bug-prone

bookkeeping work. I wasn't too eager to try that again, so I went with the other common method of implicit surface tessellation, marching cubes.

Deriving the Vertices: Marching Cubes

The idea behind marching cubes is fairly straightforward. After finding a bounding box for the entire surface, break that box into a bunch of smaller boxes to create what we'll call "cubelets." March through the cubelets (hence the name "marching" cubes), and for each one polygonize the portion of the surface inside that cubelet. The result is a polygonal approximation of the entire implicit surface.

That's just an overview, and clearly it's not much of an algorithm if it doesn't also specify how to polygonize the surface inside each cubelet. To do that, you can evaluate the implicit function at each ver-

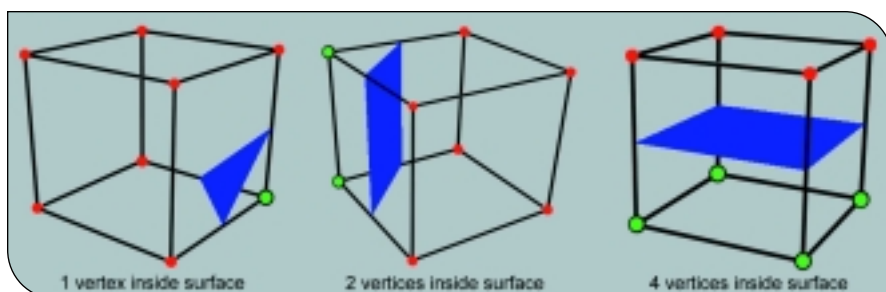


FIGURE 7. Example cubelet configurations and resulting polygonizations. Green vertices are inside the surface, red are outside.

tex, which gives you a floating-point value for each vertex. Given those point values and the isovalue of the surface we're trying to find, you can classify each vertex as either on/inside the surface or outside the surface (treating on and inside as the same thing). This is done by checking the function value at the vertex against our isovalue — if the vertex value is less than the isovalue, it's outside the surface; otherwise, it's inside the surface. This determines how you polygonize the cube: the triangulation inside the cubelet needs to separate the interior vertices from the exterior vertices. Figure 7 shows example polygonizations for cubes with one, two, and four vertices inside the surface.

Determining the polygonization that corresponds to a given configuration of vertices is somewhat time-consuming. Luckily, there aren't that many of them. Each vertex of the cube can be either in or out of the surface, which we can represent with a single bit. There are eight vertices in the cube, which brings us to a total of eight bits needed to represent the surface configuration completely. That means that there are 256 different possible cubelet configurations, which can be stored easily in a lookup table.

From the lookup table, you can determine along which cube edges you need to create vertices, and how to connect those new vertices together into triangles. What the lookup table doesn't tell you, though, is where along each edge to put a vertex. Figure 7 simply places the vertices halfway along each edge, but clearly if it were always done that way it wouldn't look very good, especially as the surface animated — vertices would snap all the way from the center of one cubelet to the center of the cubelet next to it as the surface moved through space.

LISTING 2. Pseudocode to polygonize an implicit surface inside a single cubelet.

```

polygonizeCubelet(vertices[8])
{
    unsigned char config = 0
    for n = 0 to 7
    {
        if vertex[n] is outside the surface
            set bit n of config to 1
            (otherwise leave it as 0)
    }

    for each edge in edgesNeedingVertices[config]
        linearly interpolate to find the vertex on that edge

    for each triplet of triangle indices in triIndices[config]
        create a new triangle from those indices using the
        new linearly interpolated edge vertices
    }
}

```

In practice, you linearly interpolate, or lerp, between the corners of an edge in order to find the new vertex. For an edge, e , you have endpoints v_0 and v_1 . Also, you have the value of the implicit function at each of those vertices, so v_0 has a corresponding floating-point value p_0 and v_1 has p_1 (p for potential). Then there is the iso-

value t (for threshold). To find the new vertex, do the following:

$$\text{ratio} = (p_1 - t) / (p_1 - p_0)$$

$$\text{newVertex} =$$

$$\text{ratio} * p_0 + (1 - \text{ratio}) * p_1$$

This is the simple linear interpolation between the two endpoints and the result is our new vertex. Intuitively, if p_1 is lying right on the surface (so p_1 equals t), the ratio value is 0, and the new vertex is at $0 * p_0 + 1 * p_1$, or p_1 , just as it should be. If p_0 lies directly on the surface, the ratio is 1, and the new vertex is at $1 * p_0 + 0 * p_1$, or p_0 , also as it should be.

You may be thinking to yourself, "This marching cubes stuff is terrible! The surface isn't going

to look smooth at all if each cubelet just produces a couple of triangles!" And that's true if you don't use enough cubelets. If the surface does some involved twists and turns inside a single cubelet, they'll be lost due to undersampling. It definitely takes a large number of cubelets to produce a good-looking surface, which is one of the reasons it's so important that the cubelet polygonization be very fast. Pseudocode for the whole process of polygonizing a cubelet is shown in Listing 2.

Deriving the Normals

The marching cubes pass has created vertices and faces for the mesh but if you want to light or environment-map them, you'll need normals as well. One possible way to do that is to actually differentiate the implicit function and find the gradient vector at each vertex; the gradient of the function is the surface normal at a vertex. Unfortunately, there are a few problems with that.

First, our implicit function isn't actually differentiable. Recall that I said you need to clamp the fall-

LISTING 3. Pseudocode to calculate both the potential value and color at a point given a number of molecules.

```

potential(Point p)
{
    float totalValue = 0
    Color finalColor = Black

    foreach molecule
    {
        float distance = distanceFrom(molecule center, p)
        float contribution = (1 / distance^2) - 1
        if contribution > 0
        {
            totalValue += contribution
            finalColor += contribution * molecule color
        }
    }

    // Normalize the color
    finalColor /= totalValue

    return totalValue and finalColor
}

```

off function to 0 in order to limit each molecule's range. That clamp to 0 is a first-order discontinuity: the function was pointing downwards a bit, and we clamped it to a point directly out along the horizontal axis. This means that the first derivative of the implicit function isn't continuous, and you may end up with some odd artifacts if you try to treat it as though it were.

Furthermore, even if the function were differentiable, the task of actually finding the gradients is very slow, as it would require that we walk through the molecules again for each normal. Instead, I've had good results just making a post-pass over the triangles and averaging the facet normals to generate vertex normals.

This system certainly isn't perfect. First, the polygonization can often produce triangles of very different sizes, placing some large triangles next to thin slivers. Second, because we're only interpolating the vertices linearly along cubelet edges, the vertices aren't necessarily exactly on the surface — they are just pretty close. This is because linear interpolation is only 100 percent correct when the function you're interpolating is itself linear, which our implicit function isn't. The result is that the face normals might have slight errors compared to those that have face vertices lying on the surface itself.

Averaging the face normals can therefore introduce some artifacts, and indeed the final surface usually shows a very faint rippling throughout the normals, presumably from these problems. It's not very noticeable, though. I've tried various weighting schemes, weighting the face normals so that larger faces contribute more to the vertex normals, or alternately so that smaller faces contribute more, but in the end, weighting all faces equally produced the best visual results.

Deriving the Colors

There are two parts to vertex colors. You get vertex colors from per-vertex lighting, which is straightforward since you have the vertices and normals around.

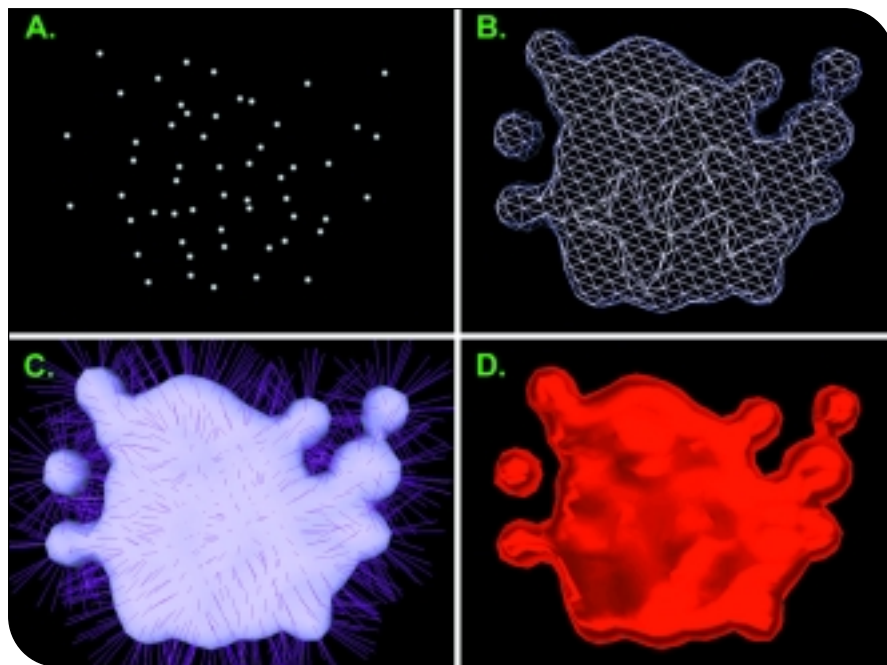


FIGURE 8. An example implicit surface. A: The molecule points. B: The polygonized surface in wireframe. C: The lit, smooth-shaded surface, with vertex normals drawn. D: The lit, smooth-shaded, environment-mapped surface.

However, there's also the question of underlying surface color — for blood, for instance, we'd want the surface to be red. So far, I've managed to get by with a single color for the entire surface. It would be possible, though, to give each molecule a color. During tessellation, whenever evaluating the implicit function, I'd track the contribution of each molecule to the implicit value and sum up each of their colors scaled by their contribution. At the end, I'd normalize the color, which produces the final color. When lerping new vertices, the colors would be lerped along with the position.

This could be useful for a variety of effects. For instance, in an area of river rapids, you could start molecules as clear and dark blue, and move them slowly towards opaque and white every time they collide with something, all the while fading them back to clear blue over time. That way, very turbulent areas would look like frothy, white rapids, whereas the calmer water would be clear and tranquil.

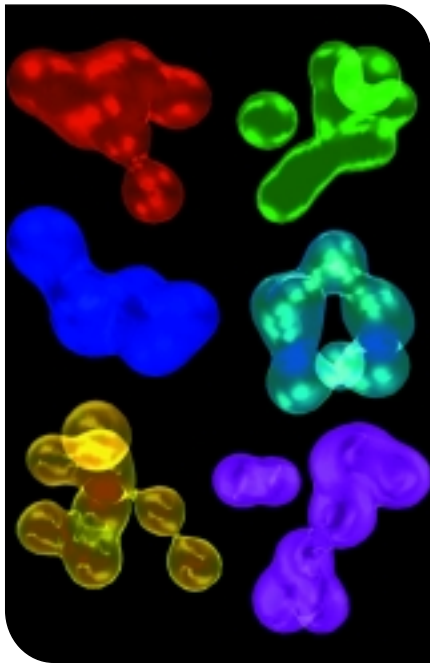
Of course, this comes at a cost. Whereas finding the normals takes $O(V+F)$ time

where V is the number of vertices and F is the number of faces, this process could potentially take $O(VM)$ time where V is the number of vertices and M is the number of molecules. This happens if all the molecules are clustered together; since every molecule contributes to every vertex, finding the color involves doing work for every molecule on a per-vertex basis. Listing 3 is a modified version of Listing 1, which also calculates interpolated molecule colors.

Deriving the Texture Coordinates

At first thought, it seems silly to think that any current game engine would ever render nontextured objects. These days, everything's textured, usually multiple times. But it's not immediately obvious how to apply a texture to an implicit surface. The vertices are generated completely dynamically, and furthermore, the surface topology may change entirely from one frame to the next.

Of course, this makes sense intuitively — we're modeling fluid. How would you



Some additional examples of implicit surfaces in action.

apply a texture to a fluid? You certainly can't shrink-wrap a texture onto a river in the same way you usually think of applying textures in modeling packages. The closest technique to standard, artist-specified texture coordinates is found in Pedersen's "Decorating Implicit Surfaces" (see For More Information). He succeeds to some extent, but the work is distinctly non-real-time and not particularly compatible with moving surfaces.

That's not to say that we can't texture implicit surfaces at all. We just can't apply textures to them in an editor and load them in with the game. We can, however, environment-map the surfaces, which uses their normal vectors to generate texture coordinates; I've had very good results using spherical environment maps to simulate static reflections. Other possibilities are projective shadows and lighting — a river flowing under a tree could have a shadow map projected down onto it, which is to say that it generates texture coordinates based on the vertex positions. There are other possibilities, too — refraction, bump mapping, and many more. Certainly, between all the possible dynamic texturing effects, slather-

ing the obligatory three or four textures (at least) should be no problem at all.

Taking a Breather

We've made a fair amount of headway in this article. At this point, you should know how to define an implicit surface using a bunch of points and the fall-off function given earlier. You should also be able to polygonize the surface with a brute-force marching cubes algorithm, brute force meaning that you sample every single cubelet in the surface's bounding box regardless of whether it contains the surface or not. You should be able to find the mesh vertices, and for each vertex find a normal and a color, and furthermore you should be able to generate an assortment of texture coordinates using OpenGL texture-coordinate generation or some other similar technique. You can also move the points around but we haven't covered the physics system yet, so the resulting animation might not look much like fluid.

Along with the physics system, other things I haven't dealt with include optimization: the brute-force marching cubes algorithm is painfully slow, as the vast majority of the cubelets it samples are empty and therefore end up wasting your time and effort. For now, a teaser for next month's article: Figure 8 shows a number of renderings of an example implicit surface.

Next month we've got a lot more ground to cover. We'll start off with a quick review of algorithm analysis theory. Armed with that, we'll go on to look at optimizations and enhancements you can make to the tessellation system to coax it to run at an acceptable speed. We'll discuss the physics system so that you can move the molecules around in a believable and aesthetically pleasing way. From there it's on to a discussion of the demo, from a development walkthrough to a brief code guide for some of the stickier parts.

When we finally finish, we'll have a fully featured demo complete with source, binaries, and stunning programmer artwork. Until then, get a head start by browsing through the publications in the

FOR MORE INFORMATION

WEB SITES

Author's web site
www.maniacal.org

Paul Bourke
www.swin.edu.au/astronomy/pbourke

ACM Digital Library
www.acm.org/dl

Organica
www.coolfun.com/html/organica.html

POV-Ray
www.povray.org

RealFlow
www.realflow.com

PAPERS

(All available at the ACM Digital Library web site)

Bloomenthal, Jules, and Keith Ferguson. "Polygonization of Non-manifold Implicit Surfaces." *Proceedings of Siggraph '95*. pp. 309–316.

Desbrun, Mathieu, and Marie-Paule Gascuel. "Animating Soft Substances with Implicit Surfaces." *Proceedings of Siggraph '95*. pp. 287–290.


Froumentin, Max, and Eric Varlet. "Dynamic Implicit Surface Tessellation." *Proceedings of the ACM Symposium on Virtual Reality Software and Technology 1997*. pp. 79–86.

Pedersen, Hans K hling. "Decorating Implicit Surfaces." *Proceedings of Siggraph '95*. pp. 291–300.

BOOKS

Cormen, T., C. Leiserson, and R. Rivest. *Introduction to Algorithms*. Cambridge, Mass.: M.I.T. Press, 1998.

Bloomenthal, Jules, and others. *Introduction to Implicit Surfaces*. San Francisco, Calif.: Morgan-Kaufmann, 1997.

References section, searching the web for more information, or even better, coding up a renderer of your own. 

STAY TUNED FOR PART 2 IN NEXT MONTH'S ISSUE.

Profiling, Data Analysis, Scalability, and Magic Numbers, Part II

Using Scalable Features and Conquering The Seven Deadly Performance Sins

Last month we discussed some of the performance issues facing AGE OF EMPIRES II: THE AGE OF KINGS (AoK). I described some of the tools that we used at Ensemble to collect that data, including Intel's VTune, NuMega's TrueTime, and our own profiling code. In this concluding article, I'll describe how to improve performance by effectively using performance tools and instrumentation. We'll also look at general types of problems we encountered as we optimized AoK which can affect any game. Then we'll wrap things up by taking a look at the last bastion of getting a game to run on the minimum platform when all else fails: scalable features.

The Seven Deadly Performance Sins

All the performance problems AoK encountered fell into one or more of seven general categories. These problems ranged from executing dead code to inefficient code and they can affect any game. Let's take a look at these categories.

1. Executing dead or superfluous code.

Over the course of a long development cycle, a lot of code-based functionality is created, changed, and/or discarded. Sometimes discarded or superseded functionality is not removed from the game and continues to be executed. While it's a waste of effort to optimize code that should be removed in the first place, it can be diffi-

cult to determine whether a few lines of code, a function, or an entire subsystem is going unused.

One feature we had envisioned for AoK was renewable resources, so natural resources such as trees would increase over time if they weren't depleted. After play-testing the game, we found that this feature would often cause a game to last indefinitely, so we eliminated it. Later, when profiling game performance, we discovered that not all of the code had been removed — the code that controlled tree regrowth appeared at the top of our profiler's function list, and we quickly removed it.

Unfortunately, superfluous code is not always so easily found, and often it's only when the code gets executed enough that you spot it on a profiling list. Such was the case with another problem also related to the trees in our game.

In our derived unit hierarchy of classes (described in last month's article), we easily added new units to the game by deriving new classes in the hierarchy. This hierarchy also is powerful in that functionality can be added or changed in a single place in the code to affect many different game units. One such change inadvertently added line-of-sight checking for trees, which is unnecessary since trees are not player-controlled. This was not an obvious performance problem and it was found only through logging data and stepping through code while trying to make the line-of-sight code faster.

2. Executing code too much. Trees, wall segments, and houses were often indicators

of general performance issues in AoK, given the large amount of them on maps — some AoK maps contain more than 15,000 trees. In order to process these units quickly, we created shortcuts in various derived functions within the unit hierarchy to avoid unnecessary unit processing. This became very complicated in some circumstances, since the computer player uses walls and houses as part of their line of sight. If it weren't for the differences between the way computer and human players used these units, the wall and house special processing would have been simpler. But the player's ability to use the buildings to scan for enemies made our AI processing simpler and more effective.

Pathing was another system that we spent a lot of time optimizing so that it wouldn't execute for too long. To do this, we capped the number of times the path-finding system could be executed to a fixed number of iterations per unit per game update. When trying to optimize a pathing system by capping its execution, you have to balance the desire to limit CPU usage with the desire to not make players think the units exhibit dumb behavior when instructed to move or attack. This forced us to tweak the game a great deal to achieve the right balance between playability and speed, but that's often the trade-off you face when optimizing a game.

We tried a variety of caps to optimize the pathfinding system, and it was determined that at five or more pathing attempts, units attempting to retarget were

AUTHOR'S BIO | Herb Marselas currently works at Ensemble Studios. He helped out on AGE OF EMPIRES II: THE AGE OF KINGS. Shhhh! Please don't tell anyone he's working on a secret 3D-engine project called [deleted]. Previously, he worked at the Intel Platform Architecture Lab where he created the IPEAK Graphics Performance Toolkit. You can reach him at hmarselas@ensemblestudios.com.



TOP. Briton Dark Age, Castle Age, and Post-Imperial Age towns. Western European building set.
BOTTOM. Viking Dark Age, Castle Age, and Post-Imperial Age towns. Eastern European building set.

the most responsive to the player. Five attempts were too many for the minimum platform, and we decided that two pathing attempts were too few based on the results of play-testing. We ultimately decided to cap the number of pathing attempts at three, once again based on our desire to balance playability with usability.

We also placed execution caps on other systems to improve performance. These included the number of pathing attempts made by a player's units, the amount of time the computer player could spend thinking during each game update, and the number of targets a unit could look for when retargeting.

3. Using inappropriate algorithms. While the pathing system in AGE OF EMPIRES was a good general purpose system, it broke down in some specific circumstances (as discussed last month). Also, there were new performance issues raised by AOK, including a larger number of units and larger maps to path across.

We could have continued to attempt to optimize the single-pathing system, but it was obvious from the work performed on

AOE that enough requirements had changed so that the algorithm could no longer stand on its own. What had been a good algorithm for AOE had become an inappropriate algorithm for AOK due to new and changing pathing requirements.

The AOE pathing system was used to path units from one general area to another over short distances in AOK. New pathing systems were added to path units quickly across the game map and to path units accurately within short distances.

Also, as part of the new pathing system, a new unit obstruction manager (see Pottinger in the For More Information section) was added for detecting unit collisions during pathing.

4. Encountering exceptional data. Built for efficiency from the start, the unit obstruction manager surprised us when it was identified by our performance profilers as one of the top problems. After reviewing the code to look for obvious (or not obvious) problems, we added instrumentation code that catalogued how units and their locations were stored within the quadtree.

With this logging code in place, we

quickly saw that the majority of units placed in the quadtree ended up being not in the leaf nodes, but higher up in the quadtree branches. We also discovered that units touching the edge of a tile were interpreted as spanning two tiles, which caused performance problems. By bumping units back onto the proper tiles, we immediately saw a 300 percent performance boost in obstruction manager performance.

This code, as is most code, was written based on assumptions about the data. Programmers assume that the data processed by a function is of a certain type and will fall within certain limits or into certain sets. When data fell outside these expectations, our algorithm — which would otherwise have performed well — was identified as a performance problem.

Some sections of the game were instrumented from the very outset of development to help diagnose data processing problems that arose frequently in those sections of code. The unit AI, for instance, contained conditional `#define` statements to log approximately 50 different sets of performance information. These performance

monitors could be used alone or in various combinations to help resolve performance issues related to data processing.

5. Inefficient memory usage. Poor performance can be caused by data structures that are not cache-line aligned, random access to main memory, using too much memory, allocating memory, and data dependencies. In AOK, memory problems could be especially severe since multiplayer games can last six hours or more, during which time tens of thousands of units can be created and destroyed.

Many data structures in performance-critical areas were compacted to fit in multiples or fractions of cache lines to improve memory access. There were also other areas that could have been improved by rearranging data in structures of arrays, or streams (see For More Information section), but this would have made the code even more complicated.

To analyze and improve the memory usage of AOK, we used a number of different tools. The first tool that was a tremendous help was the set of Windows NT performance counters, which we used to examine memory statistics quickly. The NT performance counters provided a wide array of data about an application, including processor, process, memory, and network statistics. In the case of AOK, the most important memory statistic was Private Bytes, the amount of nonshared memory allocated for the AOK process.

By sampling the memory footprint at specific intervals, we created a general picture of the game memory footprint (Figures 1a and 1b). Since the game's memory requirements are effectively the same across Windows NT and Windows 98, the NT performance counters helped us examine how memory was used during a four-player game on the minimum specified player's system. This was key to helping us determine if AOK would fit within the minimum target memory size of 32MB.

Given the minimum system game requirements (Figure 2), we estimated that a game should typically last about 45 to 60 minutes. In the four-player game example shown in Figure 1a, about 21MB of memory was allocated by the game upon start up. Thirty minutes into the game, memory usage rises to around 23MB.

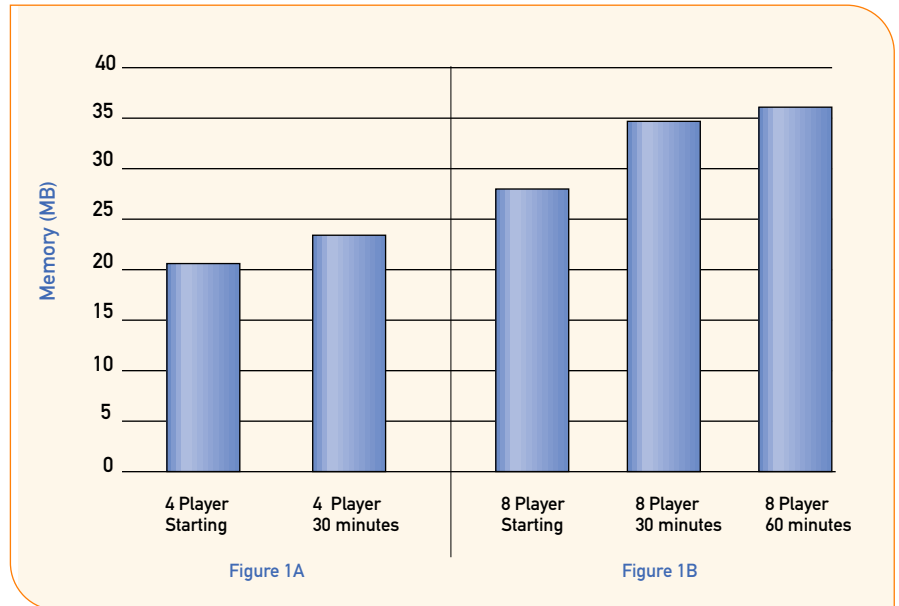


FIGURE 1A. Four-player game memory usage over time.

FIGURE 1B. Eight-player game memory usage over time.

In contrast, look at the memory footprint of the eight-player game shown in Figure 1b. The addition of more players to the game requires more memory for their data at startup, as well as more memory to support the larger game map. The amount of memory consumed continues to grow during the game as more units and buildings are created until a plateau is reached. After reaching that plateau (not shown), the memory footprint starts tapering back down. The receding memory footprint occurs as players and units are defeated.

While these high-level memory statistics from the NT performance counters are quick and useful, often it's necessary to drill down to see which specific functions are allocating memory. To get that information, we created a simple memory instrumentation system to track memory allocations (see Listing 1, pages 42–43). The memory allocation code tracked allocations and de-allocations by memory address, number of bytes requested, and file name and line number of the actual function call. It also provided a running count of the number of allocations and de-allocations, and the bytes of memory allocated in each game update loop.

The sheer number of memory allocation schemes used in AOK complicated our memory analysis. AOK uses the C++ `new`

and `delete` operators; C library `malloc`, `free`, and `calloc` functions; and Win32 `GlobalAlloc`, `GlobalFree`, `LocalAlloc`, and `LocalFree` functions. In the future, we will be actively restricting ourselves to a subset of these functions.

To reduce memory fragmentation and eliminate overhead caused by allocating and de-allocating memory, memory pooling was used in many subsystems. While this significantly increased performance, it did create problems when trying to fix bugs where code referred to recycled data.

In an attempt to improve performance further, we utilized MicroQuill's SmartHeap to manage memory in release builds. (We were unable to use it in debug builds due to incompatibilities with interactive debugging.) In the final analysis, the performance benefit of SmartHeap over the standard heap manager wasn't clear to us, due to the efforts we made to reduce and pool memory allocations.

After profiling performance and memory usage, it turned out that the most performance-limiting factor in AOK could be the Windows 95/98 virtual memory system. Unlike Windows NT/2000, Windows 95/98 doesn't require or configure a fixed-size swap file for virtual memory. To make matters worse, the swap file can grow and shrink as a program runs. An expert user

can create a swap file of fixed size, but it's not something the vast majority of users can do or should have to worry about.

AoK relies on the virtual memory system to handle the growing footprint of game data over time within the game. It also uses multiple read-only memory-mapped files to access game graphics and sounds residing in large aggregated resource files. These memory-mapped files ranged in size from 28MB to 70MB. Since the amount of virtual memory available can vary so widely on a user's Windows 95/98 system, this ended up being the number one AoK performance issue beyond our control. It should be noted that this virtual memory problem didn't effect every minimally configured system. Virtual memory problems in Windows 95/98 seemed to occur just on certain systems, even when identically configured systems performed with little or no problem.

6. Inefficient code. Rewriting inefficient code is likely the most well known performance optimization, but it was typically the last resort to fix our performance problems. In many cases, the performance problem was resolved by identifying and fixing one of the previously mentioned deadly sins.

The easiest place to attempt to improve inefficient code is with the compiler optimization settings. Due to the size of AoK, we chose to compile release builds with the default "maximize speed" setting for all modules. This may cause some code bloat (since speed is favored over size), but in general it's a good choice. We chose not to use "full optimization" since we've seen few programs that could run after using it.

Since shipping AoK we've been looking at the performance benefits of compiling with "minimize size" and then using `#pragma` (or module settings) to optimize specific hotspots for speed. This seems to be a better trade-off than just using the single speed optimization setting for everything.

In AoK we chose to use the "only `_inline`" option in Visual C++, instead of inlining "any suitable" function. This let us choose which functions to inline based on their appearance in the profile list. Inlining any suitable function would most certainly increase the code size and lead to slower performance.

Using an alternate compiler, such as Intel's C/C++ compiler, to optimize one or

FIGURE 2. AoK minimum system game specifications.

- 4 players: any combination of human and computer players
- 4-player map size
- 75-unit population cap
- 800×600 resolution
- Low-detail terrain graphics quality*

**added as part of scalability effort*

more performance-intensive modules is also another way to realize some additional performance gains. We decided against this for AoK, however, because of the risk associated with changing compilers (or even compiler versions) near the ship date.

7. Other programs. One of the greatest strengths of Microsoft Windows is its ability to preemptively run multiple programs at the same time. However, it can be a huge drawback when programs that the user is unaware of take CPU time away from a game or cause the game to lock up. For instance, during the play-testing phase of AoK's development, we received reports of problems that we couldn't reproduce on our own systems. Sometimes these issues were caused when the game entered an unstable state, but often other programs running in the background on the tester's computer caused the reported problems.

Virus scanners and other programs spontaneously running in the background while a tester was playing AoK were the most widespread cause of externally induced performance problems. Unfortunately, there's no way to easily and adequately interrogate a player's computer and warn them about potential problems that other programs can cause.

The most severe issue related to other programs involved the Windows Critical Update Notification. Play-testers sometimes reported input lock ups during game play for no apparent reason. We accidentally discovered that when AoK was in full-screen mode, the Critical Update Notification could pop up a dialog box behind AoK. This would take the focus off AoK and make it appear to players as if the game had stopped accepting input. Changing AoK to handle situations like this was relatively easy once the problem was identified. Other applications likely cause similar behavior to occur, but it's only by trial and error that these problems are identified.

Better Performance Via Scalable Features

When we had finally squeezed as much performance as we could out of AoK for the minimum platform, we



Dark Age village.

were still left with a performance deficit in the area of terrain drawing. We couldn't make the feature optional since players need to see the terrain, yet we couldn't make it any faster, either. The only alternative was to provide different implementations of terrain drawing or different levels of terrain detail.

We decided to offer three different terrain detail settings: a fast algorithm with low detail for the minimum platform, a medium detail (but slower) one for mid-range platforms, and a high detail (slower still) for high-end platforms. This allowed AOK to run on a lower minimum platform, but still give the user on the high end additional visual quality to look forward to. This was a tactic used for a number of features in AOK.

Scalable features least likely to confuse the player are those that fit directly into the context of the game, such as the number of players or the game map size. In total, there were six scalable features, four of which fit the game context. In the game, these are not called "scalable features," but "game options" (Figure 3). These options are as follows:

Number of players. The simplest scalable feature within AOK is the number of computer-controlled or human opponents or allies that a person chooses play with, which is up to eight players per game. The more players, the higher the performance required by each player's system. Up to four human or computer-controlled players can play on a minimum-specified system.

Map size. Related to the number of players is the size of the map. The map size is expressed in terms of numbers of players (for example, two-player map, four-player map, and so on). The number of players supported by a specific map size was determined by the distance that our game designers felt should be between player starting positions. But map size is independent of the number of players, so you can have a two-player game on a big eight-player map. This gives players the choice to accept our recommendations, space themselves out further, or squeeze in tighter. Based on our choice of four players for the minimum platform, the default map size is for four players.

Population limit. The unit population cap sets the maximum number of units the



Briton Wonder fortified by bombard cannon towers.

FIGURE 3. Game play and feature scalability.

NUMBER OF PLAYERS:	2 to 8, in any combination of human or computer
SIZE OF MAP:	2 to 8 player sizes and "giant" size
UNIT POPULATION CAP:	25 to 200 units per player
CIVILIZATION SETS:	Western European, Eastern European, Middle Eastern, Asian
RESOLUTION:	<ul style="list-style-type: none"> • 800×600 • 1024×768 • 1280×1024
THREE TERRAIN DETAIL MODES:	<ul style="list-style-type: none"> • High detail — multi-pass, anisotropic filtering, RGB color calculation • Medium detail — multi-pass, fast lower-quality filtering, RGB color calculation • Low detail — single pass, 8-bit color lookup

player can build during a game. By default, this value is 75, but it can range between 25 and 200 units. Again, the user does not see this as scalability, but as a tweakable option for creating the perfect game. We chose to make 75 units the default population cap because game performance on the minimum platform degrades too much at the next higher population limit.

Different artwork. In addition to these scalable game options is also an implied,

but generally unrealized, scalability in AOK's art assets. Each of the 13 civilizations in AOK is assigned to one of four sets of building art. Each building art set represents the area of the world where the civilization is from. For example, the Japanese use the Asian building art set and the Britons use the Western European building art set. There are also Eastern European and Middle Eastern art sets. These art sets not only have their own styles, each also

has different styles of buildings for each of the four evolutionary ages in AoK: Dark Age, Feudal Age, Castle Age, and Imperial Age. In other words, an Asian Dark Age house looks different from an Asian Feudal Age house, or an Asian Castle Age house.

This “upgrade” of buildings within each art set as the ages progress creates an interesting memory allocation curve. In the beginning of the game, all the players use the Dark Age version of their particular art set. As the game progresses, players advance through the ages at different rates. Since the advancement to the next Age causes the building style to change for the player, new art must be loaded and displayed. This increase in memory allocation continues until all players again reach the same age.

Assuming all players start in the Dark Age and survive to the Imperial Age, the memory allocation exhibits bell curve behavior. The worst case is when there is a player in each of the four Ages at the same time, which sometimes happens in the middle of a game. If all the players in an eight-player game select civilizations from different art sets, they use at least twice as much memory as if they had chosen civilizations from the same art set.

Display resolution. This, along with the terrain detail (which is explained in a moment), is one of two scalability options within AoK that is outside the scope of the game’s design concept. The default display resolution is 800x600, and can scale up to 1280x1024. Again, the lowest resolution was chosen for the minimum system.

Terrain detail. A terrain detail setting was introduced to reduce the amount of processor time required to draw 2D isometric terrain on slower computers by reducing the visual quality. Three levels of terrain detail are provided. The terrain highest detail setting uses multiple rendering passes, anisotropic filtering, and RGB color to bring out the best detail. The medium-detail setting replaces anisotropic filtering with a lower quality but faster filter; the low-detail setting uses flat shading and an eight-bit color lookup table similar to the terrain in the original AoE. No matter which terrain detail setting is used, the final display output only uses 256 colors.

The choice of which level of terrain detail to display is made automatically by AoK the first time it runs. Since all rendering is performed on the CPU, this decision is made quickly by using a test that gauges the CPU speed. Players can change the setting later using an in-game menu.

Unfortunately, scalable features that fall outside of the game design (such as the last two options above) are less likely to be understood by players. This lack of understanding can lead players to change settings, resulting in a negative impact on their game experience without them realizing what they did, and leave them unable to restore the original settings.

Key Lessons

After analyzing and improving the performance of AoK, our team learned some essential lessons that we hope to use to improve the quality and performance of our future products. We hope it will improve your future games, as well. These lessons are:

LISTING 1: A simple memory instrumentation system from AoK.

```

//=====================================================
// memory.h header
//=====================================================
extern "C"
{
void *mymalloc( size_t size, const char *pfilename, const long
dwline);
void myfree( void *memblock, const char *pfilename, const long
dwline);
};
//=====================================================
#ifdef _INSTRUMENTMEMORY
#define malloc DEBUG_MALLOC
#define free  DEBUG_FREE
#endif

#define DEBUG_MALLOC(size) mymalloc(size, __FILE__, __LINE__)
#define DEBUG_FREE(block) myfree(block, __FILE__, __LINE__)
//=====================================================
#ifdef _INSTRUMENTMEMORY
void MemoryInit(void);
int  MemorySave(void);
void MemoryUpdate(void);
#else
#define MemoryInit
#define MemorySave
#define MemoryUpdate
#endif
//=====================================================
// eof: memory.h
//=====================================================

//=====================================================
// memory.cpp
//=====================================================
#include <windows.h>
#include <stdio.h>
#include <io.h>
// !!! DO NOT include memory.h header file here !!!
//=====================================================
static FILE *pmemfile, *pupdatefile;
static bool binitialized = false;
//=====================================================
static DWORD gdwAllocCount;
static DWORD gdwByteCount;
static DWORD gdwDeletions;
static DWORD gdwFrameCount;
//=====================================================
void MemoryInit(void);
//=====================================================
void MemoryUpdate(void)
{
    if (pupdatefile)
    {
        fprintf(pupdatefile, "%lu\t%lu\t%lu\t%lu\n",
                gdwFrameCount, gdwAllocCount, gdwDeletions,
                gdwByteCount);
    }
}

```



```

    gdwDeletions = 0;
    gdwAllocCount = 0;
    gdwByteCount = 0;
    gdwFrameCount++;
}
} // MemoryUpdate
//=====================================================
extern "C" void *mymalloc( size_t size, const char *pfilename,
const long dwLine)
{
    RGMEMemoryEntry entry;
    gdwAllocCount++;
    gdwByteCount += size;
    void *p = malloc(size);
    if (!binitialized)
        MemoryInit();
    if (pmemfile)
        fprintf(pmemfile, "malloc\t0x%x\t%ld\t%s\t%ld\n", p, size,
pfilename, dwLine);
    return p;
} // mymalloc
//=====================================================
extern "C" void myfree( void *memblock, const char *pfilename,
const long dwLine)
{
    RGMEMemoryEntry entry;
    gdwDeletions++;
    if (!binitialized)
        MemoryInit();
    if (pmemfile)
        fprintf(pmemfile, "free\t0x%x\t%ld\n", memblock,
pfilename, dwLine);
    free(memblock);
} // myfree
//=====================================================
void MemoryInit(void)
{
    if (binitialized)
        return;
    pmemfile = fopen("c:\\memory-alloc.txt", "wb");
    pupdatefile = fopen("c:\\memory-update.txt", "wb");
    if (pmemfile)
        fputs("type\tptr\tbytes\tfilename\tline\n", p);
    if (pupdatefile)
        fputs("frame\tallocations\tdeletions\ttotal bytes\n", p);
    binitialized = true;
} // MemoryInit
//=====================================================
int MemorySave(void)
{
    fclose(pmemfile);
    fclose(pupdatefile);
    pmemfile = 0;
    pupdatefile = 0;
    return 0;
} // MemorySave
//=====================================================
// eof: memory.cpp
//=====================================================

```

- Obtain objective performance information.
- Don't leave performance issues until the code's completely done. Work to improve it throughout the course of the project.
- Don't fall prey to the Seven Deadly Performance Sins.
- Create a saved game, scenario, and/or recorded game system to capture performance workloads.
- Make performance statistics easy to collect, see, and use.
- If performance statistics aren't easy to use, they won't be used.
- Set performance targets. If you don't know how far you should go, how do you know when you've gotten there?
- Use a single memory allocation scheme.
- When all else fails, create scalable versions of features. 🐉

FOR MORE INFORMATION

Ensemble Studios

www.ensemblestudios.com

Intel VTune and C/C++ Compiler

developer.intel.com/vtune

MicroQuill HeapAgent and SmartHeap

www.microquill.com

NuMega TrueTime

www.numega.com

Performance Analysis and Tuning

- Baecker, Ron, Chris DiGiano, and Aaron Marcus. "Software Visualization for Debugging." *Communications of the ACM* (Vol. 40, No. 4): April 1997.
- Marselas, Herb. "Advanced Direct3D Performance Analysis." Microsoft Meltdown Proceedings, 1998.
- Marselas, Herb. "Don't Starve That CPU! Making the Most of Memory Bandwidth." Game Developers Conference Proceedings, 1999.
- Pottinger, Dave. "Coordinated Unit Movement." *Game Developer* (January and February 1999).
- Shanley, Tom. *Pentium Pro and Pentium II System Architecture*, 2nd ed. Colorado Springs, Colo.: Mindshare Inc., 1997.

ACKNOWLEDGEMENTS

Creating and optimizing AoK was a team effort. I'd like to thank the AoK team, and specifically the other AoK programmers for their help in getting the details of some of that effort into this paper. I'd also like to thank everyone at Ensemble Studios for reviewing this paper.

Nihilistic Software's VAMPIRE: THE MASQUERADE — REDEMPTION



GAME DATA

PUBLISHER: Activision
FULL TIME DEVELOPERS: 12
CONTRACTORS: 8
BUDGET: \$1.8 million
LENGTH OF DEVELOPMENT: 24 months
RELEASE DATE: June 2000
PLATFORMS: Hardware-accelerated PC
HARDWARE USED: Intel and AMD PCs,
 Nvidia and 3dfx 3D accelerators
SOFTWARE USED: Alias|Wavefront Maya,
 Photoshop, QERadiant,
 Visual C++
TECHNOLOGIES: 3D skinned characters,
 continuous level-of-detail,
 custom-built 3D engine,
 MP3 audio compression,
 lip syncing
LINES OF CODE: 300,000 for game,
 66,000 lines of Java for scripts.

When Nihilistic Software was founded in 1998, there were only two things we knew were certain.

The first was that we wanted to form a company with a small number of very experienced game developers. The second was that we wanted to make a killer role-playing game.

Nihilistic got started without much fanfare, just a few phone calls and e-mails. After finishing work on JEDI KNIGHT for LucasArts, the core team members had, for the most part, gone their separate ways and moved on to different teams or different companies. About eight months after JEDI KNIGHT shipped, various people on the original team began to gravitate together again, and eventually formed Nihilistic just a few exits down Highway 101 in Marin County, Calif., from our previous home.

Having moved into our new offices and bolted together a dozen desks from Ikea, our first project was to build a 3D RPG based on White Wolf's pen-and-paper franchise, Vampire: The Masquerade. Before linking up with Activision as our publisher, Nihilistic president Ray Gresko already had a rough design and story prepared for an RPG with similar themes and a dark, gothic feel. After Activision approached us about using the White Wolf license, we adapted parts of this design to fit the World of Darkness universe presented in White Wolf's collection of source books, and this became the initial design for REDEMPTION.

Because of our transition from first- and third-person action games to RPGs, we approached our first design in some unique ways. Many features that are taken for

granted in action games, such as a rich true 3D environment, 3D characters, and the ability for users to make add-ons or modifications, were reflected in our project proposal. We also adopted many conventions of the FPS genre such as free-form 3D environments, ubiquitous multiplayer support, and fast real-time pacing. To this we added the aspects of traditional role-playing games that we found most appealing: a mouse-driven point-and-click interface, character development, and a wide variety of characters, items, and environments for exploration.

Using the White Wolf license also meant that our users would have high expectations in terms of story, plot, and dialogue for the game. It's a role-playing license based heavily around dramatic storytelling, intense political struggles, and personal interaction. Fans of the license would not accept a game that was mere stat-building and gold-collecting.

In keeping with our basic philosophy, we built up a staff of 12 people over the course of the project's 24-month development cycle. The budget for the game was fairly modest by today's standards, about \$1.8 million. The budget was intentionally kept low for the benefit of both Nihilistic and our publisher. We wanted our first project to be simple and manageable, rather than compounding the complexities of starting a company by doing a huge first project. Also, we were looking to maximize the potential benefits if the game proved successful. For its part, Activision was new to the RPG market and was testing the waters with RPGs and the White Wolf license in particular, so they probably considered the venture fairly high risk as well.

AUTHOR'S BIO | *Robert Huebner is a co-founder and lead programmer at Nihilistic Software, an independent game developer located in Marin County, Calif. Prior to working on VAMPIRE: THE MASQUERADE, he contributed to several other game projects including JEDI KNIGHT: DARK FORCES 2, DESCENT, and STARCRAFT. Robert is on the advisory board for the Game Developers Conference and has presented a number of sessions there, as well as at Siggraph and E3.*



ABOVE. Professional conceptual art, such as this rendering of Alessandro Giovanni by contractor Patrick Lambert, helped the characters evolve as the art design took shape.

Development started around April 1998. When we began, we examined several engine technologies available, such as the Unreal engine and the Quake engine, but ultimately decided against licensing our engine technology. The game we envisioned, using a mouse-driven, point-and-click interface, had a lot more in common with games such as STARCRAFT than even the best first-person engines. We decided to create a new engine focused specifically on the type of game we wanted to create, and targeted 3D-accelerated hardware specifically — bypassing the tremendous amount of work required to support nonaccelerated PCs in a 3D engine. As an added benefit, the company would own the technology internally, allowing us to reuse the code base freely for future projects or license it to other developers.

What Went Right

1 • Letting the artists and designers pick their tools. With such a small team and tight budget, boosting the team's efficiency was our primary focus. If bad tools or art paths slowed down progress in the art or level design departments, we would have no chance of hitting our milestones. When we started to map the development project, the programmers gravitated toward using a package such as 3D Studio Max for both art and level design. Our argument was that doing everything in a single package would increase portability of assets between levels and art, and save the company money by licensing a single, relatively inexpensive tool. Thankfully, however, our leads in these areas strongly objected to this plan. They argued for allowing each department to use the tools that allowed them to do their work most efficiently. This single decision probably accounted for more time saved than any other.

The level designers cited QERadiant as their tool of choice, since most of them

had previously done work with id Software on QUAKE mission packs. id was generous in allowing us to license the QERadiant source code and modify it to make a tool customized to our 3D RPG environments. Because QERadiant was a finished, functional tool even before we wrote our own export module, the level designers were able to create levels for the game immediately, even before an engine existed. And since QERadiant stores its data in generic files that store brush positions, the levels were easily tweaked and re-exported as the engine began to take shape. If the level designers had spent the first six months of the project waiting for the programmers to create a level editing tool or learning how to create levels in a 3D art tool, we would not have been able to complete the more than 100 level environments in 24 months with just three designers.

On the art side, lead artist Maarten Kraaijvanger lobbied hard for the adoption of Alias/Wavefront tools for 3D art. We tried to convince him that a less expensive tool would work just as well, but in the end we decided to allow the art department to use what they felt would be the most efficient tool for the job. Since Maya was just being released for Windows NT at that time, the costs of using that toolset were not as great as we feared, and it allowed the artists to produce an

incredible number of 3D art assets for the project. During the 24 months of the project, an art department of four people produced nearly 1,500 textured models, a mind-boggling figure using any tool.

2 • Small team, one project, one room. When we started Nihilistic, we had a theory that a small number of highly experienced developers would be able to produce a title more efficiently than a larger team with fewer battle scars. In my experience, successfully delivering a game is less about what you do and more about what you choose *not* to do. Most games that ship late do so because the development team went down one or more “blind alleys” — development ideas or strategies that for whatever reason didn't pan out, and the work done in that direction is lost. As a small team on a tight budget, we could not afford to lose valuable time on these diversions. Experienced team members have the wisdom to look down a particular path and recognize when it's a likely dead end.

Developers that have shipped commercial titles also know when “enough is enough,” so to speak. There is a rampant problem in this industry of feature creep, when games end up trying to be all things to all people, and wind up taking four years to complete. Seasoned developers know that shipping a title is all about

compromise. Any title that goes out the door could always be “just a little better” and developers, even the perfectionists, are never fully satisfied with the box on the shelf. Creating a successful game that ships on time requires the discipline to draw that line and move on to the next challenge.

We also knew that we wanted an office environment where all the team members were in a single room without any walls, doors, or offices whatsoever. This didn't really seem like a radical decision — many of us got our start working for teams that operated like this — but it



Locations included both interior and exterior cityscapes, allowing dramatic situations such as this battle atop a clock tower in medieval Prague.

seems like these sorts of companies are becoming less and less common in today's industry. My first game job was working at Parallax (now Volition) software. We were eight people sitting along one wall of a narrow office space in Champaign, Ill. Even the original DARK FORCES development team was sequestered in a one-room studio in a building separate from most of the other LucasArts teams. This type of environment doesn't just foster, but rather *forces* communication between all parts of the team. For instance, a programmer can overhear a discussion between two artists about how to proceed with something and be able to jump in with an answer that will save the project days or months of work. This sort of thing happens on a daily basis; artists correct missteps by the technology team before they are made, a level designer can immediately show a bug to a programmer, and so on. Each of these incidents represents hours or days of project time saved. In an office environment with walls and doors, most of these situations would go unnoticed or unaddressed.

3 • Using Java as a scripting engine. We knew from the start that allowing the user community to edit the game was an important part of the design. After working in the first-person action-game market, we saw the benefits of supporting the user community and wanted to carry this idea over into role-playing games, where it is not the norm. A built-in scripting system makes a game engine much more extendable by fans. In JEDI KNIGHT, we created our own customized game language called COG. Creating COG took a lot of effort from the development team; several months of work went into creating the compiler, testing the generated code, and implementing the runtime kernel used to execute the scripts. The end result was worth it, but it cost a lot in terms of time and resources to pull it off (for more about COG, see my article, "Adding Languages to Game Engines," September 1997).

When starting VAMPIRE, we looked for ways to incorporate a scripting engine more easily than creating our own from scratch yet again. There were several scripting systems we examined and tested. At about that time, another game develop-



TOP. The ambitious design included parties of up to four 3D characters, each with interchangeable weapons and armor.

BOTTOM. A set of four interactive 3D head models at the bottom of the screen are skinned and animated in real time to give lifelike status for each party member.

ment company, Rebel Boat Rocker software, was getting a lot of attention for its use of Java technology. After exchanging a few e-mails with lead programmer Billy Zelsnak, we decided to give Java a try. Up to this point I knew very little of Java, and had largely dismissed it as a language suitable only for making icons dance on a web page and the like.

After a crash course in Java, we did a few simple tests incorporating it into our game engine. It passed each one with flying colors. In a matter of a few weeks, we had solved the major challenges involved in interfacing a standard, freely distributable Java virtual machine to our 3D RPG engine. From that point on, the only maintenance required was to add new native functions to the scripting language, which we did whenever we added new engine functionality that we wanted exposed to the script writers. We also trained several designers in the use of the scripting language, and they started creating the hundreds of small scripts that would eventually drive the storyline of the game.

Ever since those initial tests, I kept waiting for the other shoe to drop, so to speak. I expected to come to work one day and find out that the Java thread was chewing up 100MB of RAM or eating 50 percent of the CPU time, but amazingly, the system was trouble-free throughout development and never became a significant resource drain. If for some reason we had hit a dead end with the Java system late in the project, it would have easily taken three to four months to get back on track using a different scripting technology. In the end, the gamble paid off. We saved months of programmer time that would have otherwise been devoted to creating a scripting environment, and the result was a system significantly more efficient and robust than any we could have created ourselves.



4 ● Storyteller mode. Throughout the project, the design slowly took shape through a series of meetings that involved the entire staff. Each new design element was presented to the group and subjected to a (sometimes heated) discussion. This process of open discussion and free exchange of ideas resulted in a lot of the most interesting design aspects of the game.

It was in one of our earliest design meetings that we came up with the idea of developing the multiplayer aspect of the game not as a typical deathmatch or cooperative system, but rather to create a “storyteller” or “dungeon-master” system. The idea was inspired by the venerable text-based multi-user dungeon (MUD) games that date from a calmer time in the history of the Internet. Many of us at Nihilistic had played MUDs in college, often to the detriment of our studies. One thing that made MUDs so appealing was the ability for “wizards,” high-ranking users of the MUDs, to manipulate the game environment and create virtual adventures for the players in real time. The Vampire license from White Wolf emphasizes the role of the “storyteller,” or moderator, so we felt the time was right to take this style of play out of the college computer lab and into a commercial RPG.

Implementing the storyteller system turned out to be fairly simple from a technology standpoint. Most of the basic functionality for a storyteller game is identical to what would be required in a traditional client/server multiplayer game. The added cost was mostly in the area of design and the user interface. It took a bit of experimentation and redesign to arrive at an interface that was powerful enough to run games as a storyteller without being overly confusing to the novice player. The UI work included new interface panels with lists of objects, actors, and other resources, and a few buttons to manipulate the selected resources. Our overall design goal for the user interface was to ensure that important functionality was accessible using only the mouse, and all keyboard functionality represented only “advanced” controls such as hotkeys and shortcuts. Even though the storyteller system is something used primarily by advanced players, we wanted to preserve this design goal, which meant quite a bit of extra UI work to make a mouse-driven interface powerful enough to drive a storyteller game.

In the end, the storyteller feature ended up being one of the gems of the game design, and resonated with both the press and gamers alike. Activision made good use of the feature in their PR and marketing campaigns, and we hope the expandability and storyteller aspects of the game will give the game an increased shelf life.

5 ● Using experienced contractors. One problem with our strategy of using a small core team is that we couldn’t possibly cover all the aspects of designing a commercial game with just 12 people. Instead, we relied heavily on external contractors for certain key aspects of the game.

Sound was one area where we made use of external talent. Our colleagues from LucasArts referred Nick Peck to us, based on his excellent work on Tim Schafer’s GRIM FANDANGO. Nick ended up not only supplying us with sound effects, but also working on some of the additional voice recording and ambient loops. For our music, we teamed up with Kevin Manthei who scored the

Dark Ages portion of the game, and with Youth Engine, a local duo, for the modern-day tracks.

Even in the conceptual stages, we used external artists to help us sketch and visualize the game. Peter Chan was the lead conceptual artist for JEDI KNIGHT and had subsequently become an independent contractor. His work in the first months of the project was key in establishing the look of the game's environments. We also worked with Patrick Lambert for character concepts and he delivered incredibly detailed full-color drawings that really brought the characters to life for the modelers and animators.

Perhaps the most critical external relationship was with Oholoko, a small startup spun off from Cyclone Studios. We hired them to do our cinematic sequences that introduce the story and provide the endings. While starting the project, we met with several firms specializing in computer animation, but pretty much across the board their rates were well beyond our budgets for that part of the game. It seems that the high demand for computer animation from movies and television has driven the larger firms' prices beyond the reach of typical game budgets. By working with a smaller, less established company, we were able to get more bang for our buck in our cinematics, and the results proved to be of the highest quality.

What Went Wrong

1. Overly ambitious design. In retrospect, we were in some ways our own worst enemy. Many of the team members had wanted for some time to do a really huge, ambitious role-playing game. When we actually started the project and had a budget and schedule, we probably weren't realistic about how long RPGs typically take to develop, especially one that travels to four different cities across an 800-year timeframe. We were very reluctant to make big cuts in the design, such as cutting one of the two time periods or removing the multiplayer aspect. Because of this, we eventually had to make the decision to miss our first scheduled release date of March 2000. We also cut back on our plans to release an interactive demo some months before the game and scaled back the scope of the multiplayer beta.

Fortunately, by expanding the schedule a few months (from March to June), we were able to preserve almost all the elements from the initial design. But to accomplish this, the art and design departments really had to work above and beyond the call of duty for an extended period of time.

We did cut back a bit in the area of multiplayer by removing the ability to play through the entire single-player scenario cooperatively as a team, and instead replaced that with two smaller, custom-made multiplayer scenarios using levels and art from the single-player game. Part of this was because we did not plan properly for multiplayer when making some of the Java scripts that drive the single-player game. If the multiplayer game had been functional earlier in the schedule, the single-player game scripts might have been written from the start to be "multiplayer friendly" and we could have shipped more multiplayer content in the box.



ABOVE. Characters were created with a budget of between 1,000 and 3,000 triangles. Boss characters, such as Ahzra the Tzimisce Elder were generally the most complex.

OPPOSITE PAGE. All of the more than 100 3D characters, such as Lucretia, a Setite priestess, were modeled and animated by hand by a team of four artists using Maya.

2. Prototyping with a proprietary API. When we started developing the 3D engine for the game, which we named Nod, the 3D API landscape was quite a bit different from how it is now. We decided to use Glide as an initial prototyping API with the belief that it would be a more stable platform and avoid the complexities of supporting multiple hardware through a more general API until we had solidified the engine a bit. However, once we had a basic, functional engine running under Glide, the programmers' attentions turned toward game play and functionality rather than switching the graphics engine to a more general API such as Direct3D or OpenGL.

Because of this "if it ain't broke" mindset, we expanded our support beyond Glide fairly late in development. At the first public showing of the game at E3 in 1999, we were still basically a

Glide-only game, which meant we couldn't demonstrate the game in 32-bit modes or support some features not present in Glide at the time.

The extensive use of Glide also gave us some unrealistic performance estimates for other hardware. Since Glide allows low-level access to things like texture-memory management, we spent significant time writing our own optimized texture manager. When we switched to Direct3D, most of this work had to be discarded. Since Glide allows more flexible vertex formats than Direct3D, some of our underlying data structures needed to be changed, which meant re-exporting hundreds of levels and models. We were making low-level architectural engine changes at a stage when the engine should have been pretty much locked down. Also, because we switched late in our development schedule, we probably didn't spend as much time as we should have on compatibility testing with a wide variety of hardware. In retrospect, we should have switched to Direct3D or OpenGL several months earlier in the development schedule.

3 • Pathfinding difficulties. One problem we identified early in the development process was the problem of pathfinding. Navigation of variably-sized characters through a completely free-form 3D environment is one of the most difficult problems I've had to tackle as a game programmer. Unit navigation is hard enough when you have a flat 2D plane or restricted 3D environment, but in an environment where the level designers are free to make stairs, ramps, or any other 3D construct you can imagine, the problem becomes exponentially more difficult. My natural tendency when presented with such a sticky problem is, unfortunately, to make it good enough for the early milestone and demo builds, and then just "deal with it later." Unfortunately, "later" quickly became "now," and "now" turned into "yesterday."

We should have tackled this problem much earlier, before the levels were near completion. We should have worked with the level designers to come up with a set of restrictions for their levels, or some additional tagging in the editor to specify to the engine where characters should and



Real-time continuous level of detail allowed models to appear highly detailed in close-ups without sacrificing speed in longer shots.

should not move. Instead, the only hints from the level-design tool were "walkable" floor flags, but little or no special marking of walls, cliffs, and other pathing hazards. Since we waited too long to address the problem, better solutions such as walk boxes or walk zones would have taken too long to retrofit into the more than 100 levels already in the can. Instead, we spent weeks making small iterative fixes to the system to hide the most extreme errors and turn what was an "A" bug into a "B" or "C" level problem.

4 • Feature and data timing. This is a fairly common problem in games I've worked on, and VAMPIRE was no different. The technology team typically looks at the development schedule and schedules that entire block of time to achieve a certain feature set. Often, however, new engine features get added too late in the schedule to be utilized fully by the designers and artists. This happened several times during VAMPIRE. Some of the more interesting special effects, for example, were added only a few weeks before the data was to be locked down for final testing. Other features that we added couldn't even be implemented extensively. For

example, we added a more flexible shader language so late that only one to two percent of the surfaces in the game were able to take advantage of it. Some features that we had originally planned for the engine, like bump mapping and specular lighting, were cut completely from the initial release because there was insufficient time both to complete the feature and to create art to drive it. We softened the blow somewhat by moving some of these features to a planned patch, which would add them later if the game proved successful.

Unfortunately there are very few programming tasks that don't require some sort of artist or designer input to find their way into the finished product, so unless programmers spend the last six months of the project doing nothing but fixing bugs, some of this is inevitable. We can justify it to a degree by looking toward the likely sequel or add-on projects as a way to take advantage of some of the engine work that was underutilized in the original title.

5 • Self-restraint. As the project was drawing to a close, we found that we ended up with a bit "too much game," as someone put it. From the start, we decided to author our data for a high-



The game's story unfolds mainly via in-game cutscenes. The excellent models allowed for the creation of intricate details such as individual fingers, which helped make these scenes feel like pre-rendered cinematics.

end platform, so we'd have a good-looking game at the end of the 24-month schedule, and also because it's much easier to scale art down than up. Unfortunately, we never really started to rein in our art and design teams when we should have near the middle of the project. Instead, we continued to add more and more resources to the project, resulting in a minimum installation footprint of about 1GB.

We authored all our textures in 32-bit color and then scaled them down at load time for 16-bit cards. Our models were also extremely detailed (1,000 to 2,000 triangles each, on average) and relied on automatic level-of-detail algorithms to scale them down for slower machines. We lit our levels with relatively high light-map resolutions. All of this made the game look great on high-end systems but it meant the game was fairly taxing on low- to mid-range systems. In the end, the game just barely fit on two CD-ROMs.

We had originally planned to include both 16-bit and 32-bit versions of the game textures and allow players to choose which version to install, but after all the art was completed there was no room on the CD for more than one version.

Likewise for sounds: we wanted to include multiple quality levels but space prevented this. We actually compressed most of the voice samples with MP3 and had to remove several sounds from the game in order to fit it on two CDs.

In the end, our game looked gorgeous but had difficulty running on machines with less than 128MB of RAM — and even then, it used a fair amount of space on a swap drive. This glut of resources will also make it more difficult if we choose to port the game to a more limited console environment.

At Last, Redemption

For the first project from a new development startup, I can't imagine how things could have gone much better than they did, except perhaps if we could have avoided shipping it the same year as *DIABLO 2*. As a company, we managed to accomplish the three most important things in this business: not running out of money, not losing any team members, and actually shipping the product. Our publisher remained committed to the project throughout its life cycle, and even

increased their support as the project continued to take shape.

The course of development was amazingly smooth, with very few surprises or conflicts along the way. In this industry, you can almost bet that at some point in a two-year development cycle something traumatic will happen to either the development team or its publisher, but for us the waters were remarkably calm. About the most exciting thing to happen during development was when we lost our entire RAID server while attempting to add drivers to it, resulting in the loss of a few months' worth of archived e-mails.

Our good fortune allowed the team to focus strictly on the game and prevented distractions from outside the company. Also, keeping our company focused on just one title and resisting the frequent temptation to take on more work and more staff allowed everyone to be on the same team with little or no secondary distractions.

Hopefully, by avoiding feature creep and a four-year "death march" kind of ending to this saga, we can avoid a lot of the burnout that we have seen and often experienced on other teams. By maintaining links with both the fan community through our web board, and with the developer community at large by attending shows like GDC, E3, and Siggraph, our team was able to keep a positive attitude and high energy level throughout the schedule. We remain convinced that small development teams with a single-title focus are the best way to ship quality titles consistently, so our plans moving forward are to staff up gradually from 12 to perhaps 16 people over the next few months and embark on our next two-year ordeal a little older, a little wiser, and just a tiny bit larger. 🍷



Fighting a War Without an Army

It shall be illegal to sell, loan, or exhibit to a minor any picture photograph, drawing, sculpture, videogame, motion picture film or similar visual representation or image, book, pamphlet, magazine, printed matter, or sound recording containing explicit sexual or violent material or detailed verbal descriptions or narrative accounts of explicit sexual or violent material which predominantly appeals to prurient interests, is patently offensive to prevailing community standards, and is utterly without redeeming social importance for minors.”

Sound like something out of a repressive totalitarian state? It's not. It's an excerpt from an amendment offered on the floor of the U. S. House of Representatives last spring by House Judiciary Committee chairman Henry Hyde (R-Ill.). The amendment was decisively defeated, but left many of us in the videogame industry rather chilled.

How about this: a bill to prohibit the sale or rental of any videogame rated Mature or Adult Only to people under 17, and to prohibit the admission of people under 17 to businesses in which restricted games are shown or displayed. In other words, no one under 17 could enter Wal-Mart if a copy of *QUAKE* were on the shelf.

That's what a bill proposed in Minnesota would accomplish. The Interactive Digital Software Association and its allies barely defeated this bill in committee on a 6-6 tie vote.

Back in January, President Clinton used his State of the Union address to call on Congress to enact a bill introduced by Senator and former Republican presidential candidate John McCain (R-Ariz.) to require all entertainment industries to develop a common rating system, or else. The “or else” is that the federal government would create such a system and then subject retailers to criminal sanctions for failing to enforce these government standards.

And presently, the Federal Trade Commission is halfway through an investigation into the marketing practices of all the entertainment industries. It has issued extensive and massive document requests to nearly a dozen companies in our industry alone.

These are flush times indeed for the entertainment software business. Record U.S. sales of \$6.1 billion in 1999, new hardware systems launching in the U.S. this year and next, *Newsweek* cover stories on the Playstation 2, conferences on the industry's impact on popular culture at M.I.T., there's even a new interdisciplinary Gaming Studies program at the University of California at Irvine,

the first step toward a “major” in gaming.

But it's a time of great peril as well. The by-product of our success in establishing the industry's economic and cultural importance is greater visibility than ever before. In short, we're no longer flying below the radar — we're in the line of fire.

Now we've arrived at a point where legislation is currently pending in no fewer than six states: New York, New Jersey, Pennsylvania, Minnesota, Tennessee, and West Virginia. Last year, nine states entertained 17 different videogame violence bills.

The IDSA has met these political challenges aggressively and, to date, successfully. But I will tell you that it has not been easy and one of the most significant handicaps we face is the utter lack of involvement of many of the companies and individuals that make up our industry.

No matter how many times and how thoroughly the IDSA testifies and lobbies before Congress, there is no stronger voice than that of the constituent. Your voice as game developers does count and it is time for your voice to be heard. It doesn't matter whether you're in San Jose, Raleigh, Boston, or Austin, what's going on inside the Beltway and in far-off state legislatures will affect your business and your craft

continued on page 55

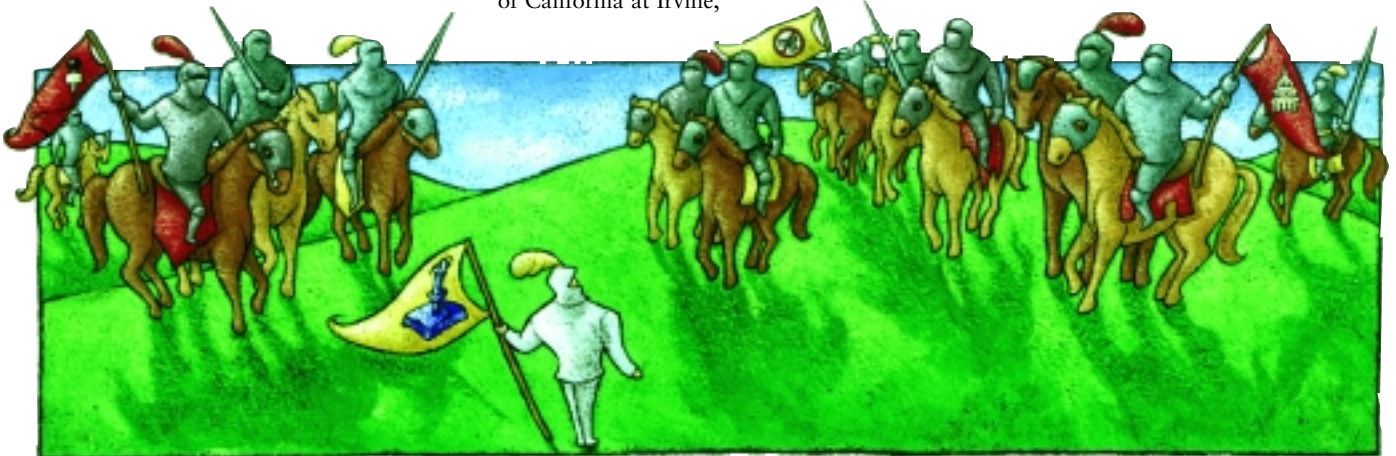


illustration by Lisa Ringnalda

continued from page 56

sooner or later. Many outside the game industry are not aware of the complexity of game development nor the industry's role in driving technology, and this includes politicians and the media. Educate them. Let them know what you do, why you do it, and why you think it's important.

From writing a letter to the editor of

RESOURCES


Interactive Digital Software Association
www.idsa.com or e-mail Jeff Woodbury at
jeff@idsa.com for information

International Game Developers Association
 (formerly the Computer Game Developers Association)
www.igda.org

your local paper to contacting your members of Congress, what may seem to be the smallest personal action will have an effect on Capitol Hill and state legislatures across the country, and go a long way toward establishing our industry as an important social, cultural, and political force in the 21st century.

You can also help this cause by making sure that your products are rated by the Entertainment Software Rating Board and that your advertisements follow the standards set out by the ESRB's Advertising Review Council. These programs ensure that parents have the information they need to make appropriate game purchasing decisions, and that advertisements are designed and published or aired responsi-

bly. Participating in these programs sends the message out that your company, like hundreds of others in the industry, is acting in the best interests of consumers.

All of you have a stake in the success of this grassroots campaign. Without your involvement, we run the risk of continuing to be seen as a peripheral part of the entertainment industry, not the increasingly dominant player we actually are. I know this will take some time and effort, but it will be well-spent and pay long-term dividends. 

AUTHOR'S BIO | *Mr. Lowenstein has served as president of the Interactive Digital Software Association since its creation in June 1994.*

ADVERTISER INDEX

COMPANY NAME	PAGE	COMPANY NAME	PAGE	COMPANY NAME	PAGE
Adaboy	17	Hewlett-Packard	23	Numerical Design Ltd.	C3
AICS	53	IBooks.com	12	RAD Game Tools	C4
Ascension Technology	8	LIPSinc	11	Rainbow Studios	53
Cinram	54	Midway	52	Seneca College	54
Conitec	55	Motek	3	SN Systems	21
Dice.com	52	Multigen-Paradigm	4	Vancouver Film School	53
Havok.com	C2	Newtek	18	Viewpoint Digital	7