GAME DEVELOPER MAGAZINE

FEBRUARY 2000

UBM Tech

Back

Local Maxima

"A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines."
— Ralph Waldo Emerson

It's conventional wisdom in the game industry that desktop personal computers have to become more like videogame consoles or they will cease to be a viable platform for which to develop games.

I'm not so sure. No, I don't dispute that PC game publishers spend tons of money on technical support because PC games rarely work for consumers out of the box. Similarly, I don't dispute that selling seven million copies of a console game in its first weekend looks very nice compared with selling 70,000 copies in a PC game's lifetime. And who would dispute the allure of developing games for a piece of hardware that's guaranteed to be there when somebody runs your game?

Still, I'm going to miss the current PC architecture if it truly does go the way of the dodo. The best word I can come up with to describe the trait I'll miss is "wackiness." The PC is a wonderfully wacky medium, and I think even the console industry benefits from its wackiness.

What do I mean by "wacky"? Take the original Nvidia NV1 chip, one of the first attempts at consumer-level 3D graphics in hardware. Now that was wacky. The NV1, for you young, TNT-spoiled whippersnappers, rasterized quadratic surfaces directly. Who cares whether quadratic surfaces don't mesh well with any known modeling paradigm in this universe? Who cares whether anybody ever actually used this hardware to make a game? It certainly caused a ruckus and made for some very interesting conversations among game developers.

Or how about Aureal's 3D sound API that expected you to upload your world geometry? Definitely wacky. Intel's MMX instructions? Wacky. D3D execute buffers? Beyond wacky. There are infinite examples.

I actually think the PC 3D hardware industry is about to go through a renaissance of wackiness after the last couple years of relatively boring fill-rate and throughput competition. Now that everybody's implemented the same classic SGI-inspired pipeline, no one is quite sure what features to implement next. SGI's not leading the pack anymore on features, and so this next generation of 3D hardware is going to go every which way as vendors throw things against the wall to see what will stick.

This wackiness and volatility is not necessarily a bad thing. For example, when you're trying to maximize something using mathematical optimization, you don't necessarily always want to keep going in the obvious direction, the one that constantly increases your objective. If you do this, you might get stuck at the top of a foothill when the true peaks are a bit farther away. The top of this foothill is a local maximum, where every direction from here goes downhill and reduces our objective, even though we're not at the global maximum — our true goal.

This mathematical concept carries over to the game industry; these wacky ideas keep us from getting stuck. Their randomness bounces us around a bit, shakes things up, and maybe gives somebody an insight into a completely different way of doing things.

Consoles don't have this level of wackiness, at least when you consider the rate at which they change. Sure, consoles like the Sega Saturn are plenty wacky when they're released, but a few months later everybody's used to it, and three years later it's downright boring.

There's an intrinsic tie between the pace of technical innovation and the stability of a platform, and so it stands to reason that if we give up our instability, we'll have to reduce our pace of innovation. Don't get me wrong, I think consoles are great, especially for their users. It's probably inevitable that the PC will evolve in that direction. However, I will definitely miss the technological wackiness of the PC. ■

Chris Hecker
Editor-At-Large

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@mfi.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Jeff Lander *jeffl@darwin3d.com*
Mel Guymon *mel@infinexus.com*
Omid Rahmat *omid@compuserve.com*
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 e: *jorvik@mfi.com* t: 415.905.2156
Publisher Relations Manager
Ayrien Houchin e: *ahouchin@mfi.com* t: 415.905.2788
Account Representative, Silicon Valley
Mike Colligan e: *mike@mfgame.com* t: 415.356.3486
Account Executive, Northern California
Dan Nopar e: *nopar@mfgame.com* t: 415.356.3406
Account Executive, Western Region
Darrielle Sadle e: *dsadle@mfi.com* t: 415.905.2182
Account Executive, Eastern Region
Afton Thatcher e: *athatcher@mfi.com* t: 415.905.2323

ADVERTISING PRODUCTION

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

MILLER FREEMAN GAME GROUP MARKETING

Marketing Manager Susan McDonald

CIRCULATION

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*

Miller Freeman
A United News & Media publication

CEO/Miller Freeman Global Tony Tillin
Chairman/Miller Freeman Inc. Marshall W. Freeman
President & CEO Donald A. Pazour
Executive Vice President/CFO Ed Pinedo
Executive Vice Presidents Darrell Denny, John Pearson, Galen Poss
Group President/Specialized Technologies Regina Ridley
Sr. Vice President/Creative Technologies KoAnn Vikören
Sr. Vice President/Human Resources Macy Fecto

BPA INTERNATIONAL

Back

BIT Blasts

News from the World of Game Development

New Products

by Daniel Huebner

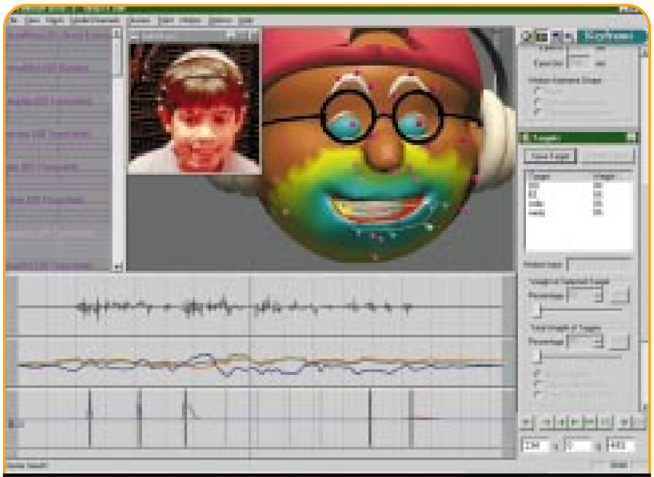
You Look Familiar...

FAMOUS TECHNOLOGIES has shipped Famous Faces 1.5, adding to the influx of new facial animation software for 3D animators. Unveiled at Siggraph 99, the package a is stand-alone system that aims to accelerate character animation workflow by allowing animators to extract 3D facial motion from .AVI files and NTSC video. Designed for use in the film, broadcast, games, and location-based entertainment industries, Famous Faces uses an actor's performance to animate dialog and facial expressions quickly, enabling animators to create more lifelike and convincing facial animations.

Among the new features for version 1.5 is multiple input support, allowing for the use of motion capture, video, puppetry devices, voice recognition, or a combination of these methods. The system has also added support for real-time game animation, an alternative to keyframe animation. Open motion-capture channels provide easy integration with common 2D and 3D mo-cap sources including Motion Analysis and Vicon.

Famous Faces 1.5 is available as a stand-alone for Windows NT, and also includes plug-in support for most major animation software, including Lightwave 3D, Softimage, Maya, 3D Studio Max, and Filmbox. It is priced at $4,990.

■ Famous Technologies
San Francisco, Calif.
(415) 835-9445
http://www.famoustech.com



Famous Faces lets animators work from existing mo-cap or video to create lifelike facial animations.

Sew It Up

REALVIZ is serving up another image processing technology designed to make life easier for 3D modelers and special-effects professionals. Stitcher's task is to combine multiple vertically- and horizontally-overlapping 2D images seamlessly for fast and simple creation of wide-angle and panoramic images. Those images can then be exported into a variety of compositing and 3D modeling packages, including Realviz's own Imagemodeler.

The minds behind Stitcher see it as an alternative to costly and time-consuming location shoots for the production of large-scale background scenes. Panoramic images up to 360 degrees by 360 degrees can be created and exported to build photo-realistic matte paintings or environment maps. Panoramas are deblurred, color-blended, and warped so they won't require additional editing, and Stitcher also provides its own set of flexible production tools.

Stitcher is available for Windows NT at a suggested retail price of $2,000 and an IRIX version is in the works for release by the end of the first quarter. Realviz also offers a pay-per-use licensing plan, an appealing alternative for small budgets that only need the software on a limited basis.

■ Realviz S.A.
Sophia Antipolis, France
+33 (4) 92-38-84-60
http://www.realviz.com

Putting Sound In Motion

HUMAN MACHINE INTERFACES has announced the Inmotion 5.1 Surround Producer, a professional application designed to let users create true multi-channel audio mixed for 5.1, 6.1, 7.1, or any other desired speaker layout, without affecting the quality or tone of the original audio stream. Inmotion's interface works by allowing users simply to draw the path the sound should follow to a set of virtual speakers.

Inmotion's environmental modeling capabilities include Doppler motion effects, air absorption, occlusion modeling, reverberation, and distance modeling. The system also allows for real-time updating of all effects parameters without audible clicks or pops. Inmotion supports audio input and output in standard .WAV and .AIFF file formats for easy integration, and supports audio sample rates of 11, 22, 44, 48, and 96KHz.

The Inmotion 5.1 Surround Producer is available for Windows 95/98/NT at an introductory price of $995.

■ Human Machine Interfaces Inc.
Eugene, Ore.
(541) 687-6509
http://www.humanmachine.com

Industry Watch

by Daniel Huebner

SONY WAGES WORKSTATION BATTLE. In an interview with Nikkei Electronics, Sony Computer Entertainment CEO Ken Kutaragi laid out Sony's plans for entering the workstation market. Sony hopes to use the development of workstation technology to drive the development of the Playstation 3 and other future products. "I don't think that the workstation business will be able to turn a profit, but we will supply the funds for development work there from the game machine side of the company," said Kutaragi.

Sony is expecting that workstation competitors such as SGI will be unable to keep up with Sony's technological pace in the future. "For 2000, we are preparing workstations with capabilities ten times better than the development tools currently available — and in 2002, they will be a hundred times better," explained Kutaragi. Sony enjoys the luxury of being able to afford the investment in semiconductor manufacturing technology on the scale necessary to maintain that kind of pace, and Kutaragi went so far as to claim that most companies would go out of business trying to keep up with Sony. Kutaragi clearly isn't hedging his bets, asserting, "SGI doesn't have enough strength remaining to go on competing at the very cutting edge of technology."

Sony chief executive Kutaragi: not messing around.

LOSSES INCREASING, SEGA RESTRUCTURES. Citing the cost of promoting its products overseas, Sega of Japan announced big losses for the first half of its fiscal year. Despite increasing sales by 25 percent and reaching the one-million mark for North American Dreamcast sales well before Christmas, Sega's losses totaled $182 million. The company recorded a profit of $11.5 million in the same period last year.

Sega also announced a major restructuring that will focus the company more on Internet gaming. "We are a believer in the Internet, so we will focus on entertainment on networks," said Sega president Shoichiro Irimajiri. "We are now aiming to provide our services on the Net...to become a network entertainment kingdom." Sega plans to list its Internet-related business in Japan and the U.S. at the beginning of its next fiscal year in April 2000.

Sega will also split its research-and-development, home-game, and arcade-game units into separate companies as it looks for ways to focus better on Internet development. Included in the planned listings will be ISOA Corp., the joint-venture company created to handle online services for the Dreamcast.

In addition, Sega will set up more than ten new spin-offs from its development business. Sega of America's senior vice president of marketing Peter Moore welcomed the announcement, saying that Irimajiri's statements reinforce Sega's commitment to expanding upon the Dreamcast's forward-thinking gaming and network functionality.

3DFX NAMES NEW CEO. The position of president and CEO at 3dfx, made available by Greg Ballard's departure in October, was filled when the company's board of directors named Dr. Alex Luepp to the top position. Luepp is a 25-year veteran of the semiconductor industry and has served as a member of the 3dfx board since October 1998. Gordon Campbell, 3dfx's chairman, cited Luepp's technology expertise, leadership abilities, and strong understanding of the semiconductor industry as contributing factors to the board's choice. Luepp joins 3dfx from Chip Express Corp., having previously spent 12 years with Siemens Microelectronics, including six as chairman and CEO.

EA PARTNERS WITH AOL. Electronic Arts has entered into a five-year agreement to be the exclusive provider of games content on America Online's Games Channel, as well as for providing game content for additional AOL properties including AOL.com, Netcenter, and Compuserve. Electronic Arts will create new games exclusively for the America Online Games Channel and also take advantage of AOL's highly trafficked venue to showcase existing EA titles. AOL's Games Channel will be relaunched in the summer of 2000 with new content ranging from card games to massively-multiplayer online worlds. EA hopes that by adding to its own company web site, EA.com, its strategic relationship with AOL will enable EA to reach millions of new consumers.

The same day EA announced its deal with AOL, the company also acquired Kesmai Corp., based in Charlottesville, Va. Formerly a subsidiary of News Corp., Kesmai is a developer and publisher of multiplayer online entertainment and a provider of content for AOL's Game Channel. ■

UPCOMING EVENTS CALENDAR

Linux World Conference & Expo

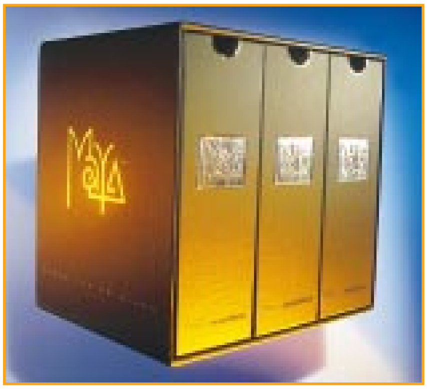
JACOB J. JAVITS CONVENTION CENTER
New York, N.Y.
February 1–4, 2000
Conference: $225–$695
http://www.linuxworldexpo.com

Game Developers Conference 2000

SAN JOSE CONVENTION CENTER
San Jose, Calif.
March 8–12, 2000
Cost: $200 and up
(early-bird discounts available)
http://www.gdconf.com

American Association for Artificial Intelligence Spring Symposia

STANFORD UNIVERSITY
Stanford, Calif.
March 20–22, 2000
Cost: $280 for nonmembers
(student rates also available)
http://www.aaai.org



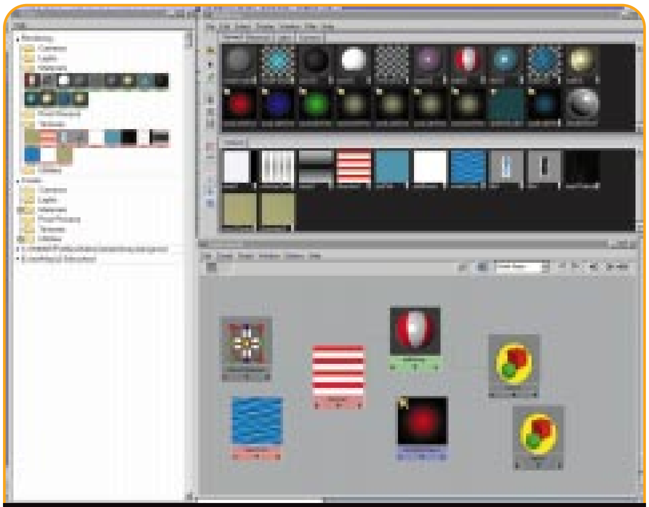
Alias|Wavefront's Maya 2.5

by Mel Guymon

Alias|Wavefront's formidable Maya software has been garnering support in the games and feature film industries ever since its release in early 1998. With its open, script-based architecture, powerful animation and rendering tools, and constantly evolving feature set, Maya seems eminently suited to the task of game development. It beat out some stiff competition to pick up a *Game Developer* Front Line Award last year, and it should come as no surprise that an ever-increasing number of developers have been choosing Maya as their weapon of choice for game development. The latest version of the software, Maya 2.5, adds some real improvements to an already strong package.

FIRST IMPRESSIONS. It was apparent immediately that this latest version of Maya has come a long way since its initial release. Over the course of the last two upgrades, there have been fixes and upgrades in almost every category. The interface has been streamlined and augmented, adding functionality and removing problematic clutter. Polygonal modeling is much easier in 2.5, and most of the functionality available in the lower-end programs has finally been added. Most importantly, the Artisan module is now fully functional with polygonal geometry. Attaching a skinned character to a skeletal hierarchy is extremely intuitive, and a new modeling method dubbed Subdivision Surfaces has been added. However, the most impressive new feature is the much-anticipated Paint Effects module. Combining the power of Maya's particle effects system with the intuitive interface of Artisan, Paint Effects will very likely revolutionize the workflow for creating 3D content. But we'll get to that later.

INTERFACE. One common criticism Maya has faced has been the number of windows and panels the user must navigate. It's ironic, then, that of the many interface improvements of Maya 2.5, the one that stuck out was the addition of the Hypershade and Visor windows in the rendering module. The Hypershade window combines the functionality of the Multilister and the Hypergraph window into one panel, allowing artists to manage the renderable nodes (materials, objects, and lights) with ease. The Visor window is simply a file browser that facilitates a drag-and-drop workflow for image files and other graphical elements. The combination of the two new panels made texturing and materials-management a snap, and the drag-and-drop functionality made creating and applying the textures extremely intuitive. Although the Multilister panel is still included for those die-hard Power Animator fans, I found that now I could do without it entirely.



The Multilister and two new windows, the Visor (left) and the Hypershade (bottom).

Another huge improvement in general workflow is the addition of an Interactive Photorealistic Renderer, or IPR. The IPR window is a fully rendered version of the scene that updates nearly in real time. Once the artist performs an IPR render, a raster file is generated which stores the results of the rendering calculations for the scene. Subsequent changes to the rendering nodes, such as lighting values or texture and material changes, are automatically updated in the rendered image. However, since most of the transform and lighting calculations have already been stored in the raster file, the updates take far less time to generate than re-rendering the entire scene, and only the affected pixels are updated. Even on my relatively low-end test machine (a 500MHz Pentium II with 256MB RAM and a Diamond FireGL card), the technique was extremely fast. Despite the limitations of my system, I found that the IPR window updated in near-real time (one to two seconds per change) changes I made to the textures and lighting in the scene. For large scenes, selecting a subregion within the IPR image can greatly increase the update speed. With Maya 2.5, IPR is now fully multi-threaded and the renderer will make use of all available processors.

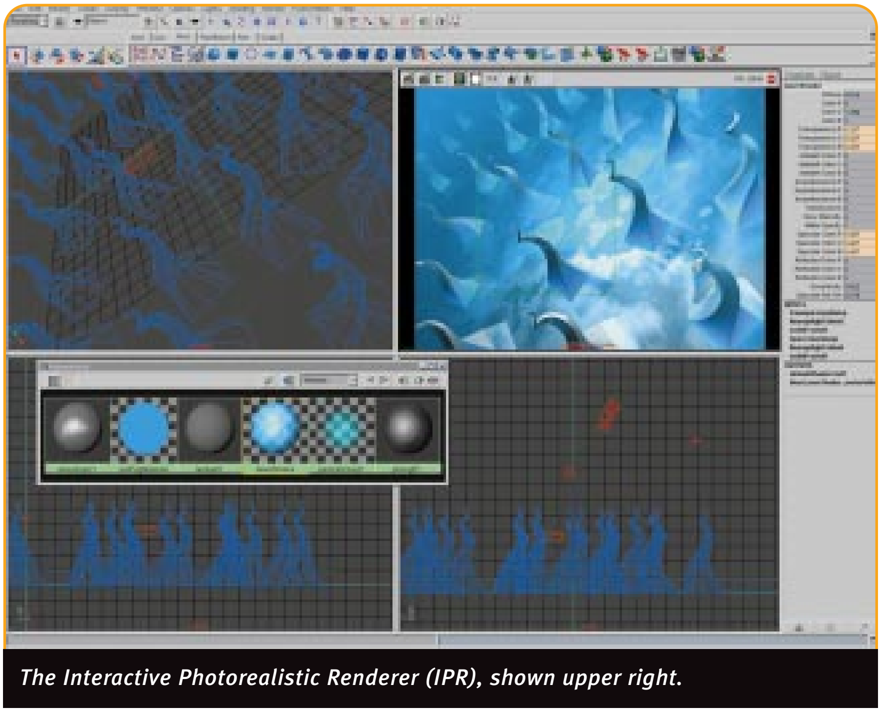
ANIMATION. Aside from some interface adjustments in the Graph Editor and Dopesheet, the animation features in Maya haven't changed much, not that they needed it. Maya's animation toolbox includes a robust IK system with multiple solvers (including an intuitive, easily-adjustable spline-based solver), a fully interactive set of constraints, and a tool for creating what are termed "set-driven keys," which is basically an extremely fast graphical method for creating complex expression-based animations. With this arsenal of functionality, it would be hard to find a more capable character animation tool.

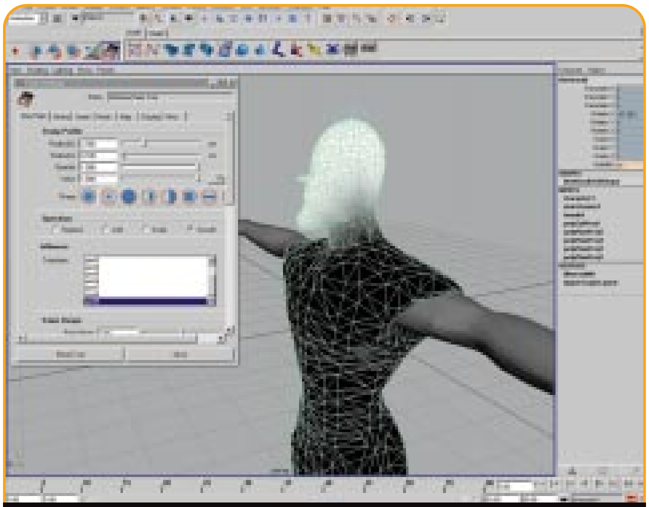
Mel Guymon has been animating in the gaming industry for several years. When he's not at his desk pushing polygons, he can usually be found at the local Barnes and Noble, slumming for reference materials. Mel can be reached at mel@infinexus.com.

One of the few updates to the animation module is the addition of what are called "Breakdown Keys." Breakdowns are a special type of key which maintain the proportional relationships in time with adjacent keys. This nonlinear editing functionality allows individual characters within a scene to have the timing on their animations adjusted proportionately, independent of global time. This is especially useful if a certain frame of the animation is required to synch up with an event in the scene, for example when two characters are interacting with one another.

MODELING. Several improvements to the polygonal modeling tools have been made, although much of this functionality is what would be considered "standard" on other, less-expensive packages. Some of the more obvious upgrades include edge flipping, polygonal object mirroring, polygonal object cleanup (removal and optimization of unwanted geometry, such as zero-area faces or zero-length edges), and a more efficient NURBS-to-polygons converter (NURBS to polygons assigns the same texture to the new polygonal object, and bakes the NURBS UV values onto the corresponding polygon vertices).

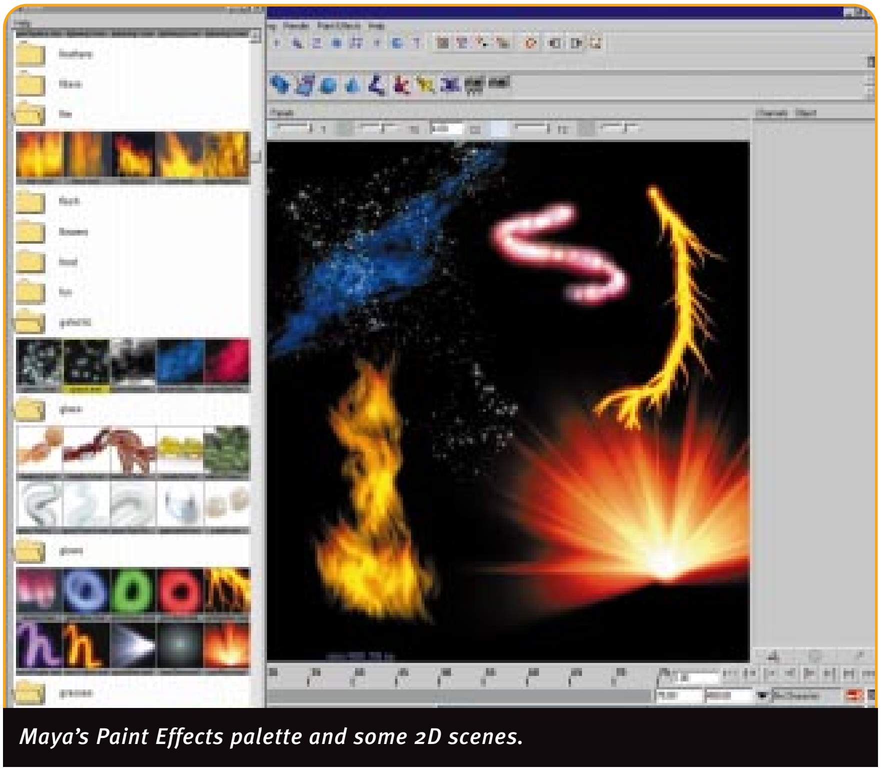
By far the most important upgrade is the ability to use Artisan tools to select polygonal components, apply color, and sculpt polygonal models, mitigating the necessity of modeling first in NURBS to gain the benefit of the Artisan functionality. Artisan has been further enhanced by the ability to paint attribute values on surfaces (for attributes that have been identified as "paintable") and by the ability to paint vertex weights on a skinned skeletal model. The latter of these two improvements, in conjunction with an easily-managed smooth-skinned weighting technique, was, in the minds of most animators, one of the shortcomings of Maya's earlier incarnations. With the latest release, I found that applying a skeleton to a solid-skinned model and weighting the vertices effectively was exceptionally easy. Joint areas that are usually problematic, such as shoulders and neck bones, animate and deform correctly with only a few minor adjustments. And the graphical representation of the weighting values is intuitive and easy to work with.


The Interactive Photorealistic Renderer (IPR), shown upper right.


Smooth-skinned weighting techniques make for easy and accurate deformation and animation.

GAME-SPECIFIC FEATURES. Much is being made of Maya's ability to serve as the complete game development package, and many of the game-specific upgrades are focused on this. For instance, the latest version includes a Level of Detail interface for setting up LOD distances within a Maya scene, and a camera "fly mode," for flying through the user-created levels in much the same way that a player would navigate them. While these features are certainly useful, what I found impressive were the IK bone handles and single chain solvers which are included in Maya 2.5. The use of IK handles (or end effectors, as they are termed in other packages) has long been a mainstay of hand-generated IK animation. The source code for this functionality has been included, so that developers can replicate this lightweight IK system into their run-time engines. The intent is that the notion of an IK handle can be extrapolated into a game engine, allowing programmers to modify existing animations procedurally on the fly. The potential uses for these are legion, for example to keep a character's feet planted on uneven ground during a run cycle or to "aim" a reaching animation when a character moves to pick something up in the world. Given that procedural IK is presently a pretty hot topic among developers, this would seem to be an ideal implementation.

PAINT EFFECTS. When the folks at Alias|Wavefront first released Maya's Artisan tool, they began to define a new paradigm for digital modeling. The notion of using a sculpting tool that conforms and flows over the surface of a 3D object seems, like most great ideas,



Maya's Paint Effects palette and some 2D scenes.

to have been obvious in hindsight. Now, nothing else seems as intuitive. With Paint Effects, Maya's developers have taken the next logical step and applied this paradigm to the particle effects system. In doing so, they've come up with a powerful and unique method for creating both 2D and 3D content, one that may well revolutionize the way we work.

Basically, Paint Effects allows artists to paint brush strokes and particle effects quickly and easily on a 2D canvas, on 3D geometry, or between 3D geometry. The workflow is simple. The artist selects from a preset list of objects, or brushes, and commences painting these objects into the scene or onto the 2D canvas. With each click of the mouse, a spline, or "stroke," is created, and it is these strokes that control the position and size of the objects created. The Paint Effects brush strokes can be painted onto virtually any surface, and splines that were not created with Paint Effects can be assigned to be emitters. The objects that are painted into the scene are actually clusters of particles, which, at render time, get replaced by procedurally-generated geometry (called stamps). These static particle systems are based on splines that exist in 3D space, and subsequently have depth and breadth. As such, they will render properly from any angle.

In the 2D Canvas mode, Paint Effects can be used as a traditional paint program to paint images, or to generate automatically repeatable textures. The similarity to a traditional painting program ends there, however, since with a single brush stroke, you can paint entire particle effects and complex images on your 2D canvas. In the 3D scene, the same brush stroke can create these objects in full 3D. As one of the Maya artists put it, "Imagine being able to paint an orchard in your scene where the painted trees exist as objects that your characters can move around. You can also apply dynamic forces to the effects you paint in your scenes and animate the display and movement of the effects. For example, you can make plants grow, make long hair blow in the wind, or make a river flow."

From a game developer's point of view, when you consider the amount of work that goes into generating pre-rendered movies and interface screens, as well as textures and special effects for real-time 3D, the timesaving aspects of Paint Effects are potentially tremendous (painting on the 2D canvas even generates the a correct alpha channel for the texture). This alone is worth the price of admission into the Maya family.

THE FINAL WORD. Although the interface still seems a bit overbearing at times (let's face it, there are a lot of menus and windows in this software), I found it hard to find fault with Maya 2.5. Most of the functionality that was oddly missing in the early releases has finally made it into the package, and the solid animation features make it a sound choice for any character-based project. Regardless, whether or not you are a dedicated Maya development house, with the Paint Effects module thrown into the package it would be difficult for anyone to argue reasonably against having at least one suite of Maya in-house. ■

Acknowledgements

Special thanks to Peter Whiteside, Irene Grubb, Matt Dougan, and Mike Aquino.

Maya 2.5: ★★★★½

Alias|Wavefront
Toronto, Ontario, Canada
(416) 362-9181
http://www.aw.sgi.com

Prices:
Maya Builder: $2,995
Maya Complete: $7,500
Maya Unlimited: $16,000

System Requirements:
See the qualification charts for Maya NT and IRIX at http://www.aliaswavefront.com/pages/home/pages/support/pages/qualification_charts/index.html.

Pros:

1. Solid character animation toolset.
2. Excellent particle and special-effects systems.
3. Exceedingly customizable script-based user interface.

Cons:

1. Lots of interface to wade through.
2. Still a bit on the expensive side.
3. Not enough flexibility with other file formats.

Competitors:

Avid Softimage 3D 3.8
http://www.softimage.com

Discreet 3D Studio Max 3
http://www.ktx.com

Newtek Lightwave 6
http://www.newtek.com

Nichimen Mirai
http://www.nichimen.com

Under the Shade of the Rendering Tree

The goal of computer graphics has always been to create increasingly realistic images. Faster processors and more sophisticated rendering techniques have allowed computer artists to create scenes that come very close to simulating reality. In particular, computer graphics are good at rendering architectural scenes. Raytracing and radiosity renderings often fool me into believing I am seeing actual photographs. Even in the real-time game market, techniques such as multi-pass rendering and precomputed lighting have enabled game players to run around in a world complete with reflections, shadows, dynamic lights, and an impressive amount of texture detail.

As I write this, Quake 3: Arena has been unleashed upon the world. This impressive game takes real-time rendering technology to a new high. Once again, the graphics capabilities in id's latest offering stretch real-time rendering to its limit, bringing even the latest "graphics processing units" to their knees. Quake 3 also marks the industry debut of programmable "shaders," which are used for describing the look of a real-time rendered image.

A shader is a form of programming language that describes the look of a particular surface in a rendered world. In its most abstract sense, it is a function that is given a series of properties and then returns the color of the light leaving any position on the shaded surface. Normally, the properties given to a shader include such things as the lights in the scene, the color of the surface, and some measure of the roughness of the surface.

Game programmers and artists don't normally think of the rendered world in these terms. However, even the most basic 3D rendered scene can be described as a collection of surface properties and light interactions. A texture map that is applied to a 3D polygon simply describes the color of the light that leaves that polygon at any point on its surface. Likewise, the Gouraud shading model is a series of parameters that controls the interaction of the lights in the scene with the color and roughness of the polygon surface. The power of a shader language, however, goes way beyond what we have traditionally done with real-time 3D rendering. Since a shader describes the color leaving the surface of a polygon, it can be used to generate a complex pattern of colors without texture maps.

Most of you are familiar with procedural textures. This is a technique whereby a texture map is created by some form of mathematical formula instead of being drawn in an art package. Procedural textures are commonly used for patterns such as noise (like TV "snow"), lava, water, marble, or fire. Unreal implemented procedural texture techniques for several effects used throughout its environments. This allowed its designers to have a nearly unlimited variety of certain types of textures without having to store all those bitmaps on the game CD. However, the textures still needed to be generated in order to load them onto the 3D card for rendering.

Wouldn't it be nice if those textures never had to be generated at all? What if I could simply upload a small program that handled all my procedural textures? Then all I would need to provide to the rendering hardware would be a few variable settings for each different material. Sounds kind of futuristic, right?

It is not as far out as you think. Shading languages have been around for quite a while. The first, and still most commonly used, is Renderman. First described in the late 1980s, this rendering language has been used to create some of the most memorable computer graphics scenes of all time, including the recent movie *Toy Story 2*. While it may seem that we are a long way from creating scenes this complex for real-time games, you may be surprised.

Listing 1 describes a Renderman shader that creates a checkerboard pattern on a surface. The shader takes three float variables and two colors and creates a checkerboard of any size and frequency. This is done without any texture map. For a checkerboard, this may not seem very impressive; however, it's the idea of controlling the look of an individual pixel on an individual surface that makes Renderman so powerful. A shader doesn't need to be as simple as a checkerboard. Shaders can be used to create all kinds of surfaces, everything from highly-detailed wood, marble, and fire to even a moldy cue ball (my favorite Renderman shader).

A closer examination of the checkerboard shader reveals that the only other thing the shader really

When not ditching work to catch the latest animated feature film, Jeff can be found at Darwin 3D trying to convince clients that things can't look any better. Tell him how wrong he is at jeffl@darwin3d.com.

needs to know about is the position of the view and the lights in the scene. Interestingly, the new generation of 3D graphics hardware such as Nvidia's GeForce 256 keeps these positions in hardware already. I can't help but think that the hardware manufacturers are thinking of the implications in the same way that I am. I don't know how long it will take, but I am going to dust off my *Renderman Companion* and start thinking about how to integrate programmable shaders into my art production pathways. Since I can't really envision many artists learning to program Renderman, I think there are going to be a lot of tools that will need to be created. However, until I get my ultimate shader language written, I am stuck with the traditional texturing and lighting methods to get the results I want.

Welcome to Toon Town

I have lamented before in this column that creating 3D characters is very difficult. I can take some comfort from the fact that even Pixar, with its terrific Renderman shading system and all the money and talent possible, has trouble getting human characters right. They have hit upon one of the great ironies of computer graphics. When rendering 3D environments, the technology has enabled increasingly realistic final images. With each advance in modeling or lighting, the images take a step closer to what we see around us in the real world.

With human characters, on the other hand, the story is entirely different. In my experience, as a 3D computer-generated human is rendered in an increasingly realistic manner, it paradoxically looks increasingly strange to viewers. They can't really say why it looks odd, just that it's not quite right. This is especially noticeable when the texture maps for the characters' faces are created from photographs of real people.

Particularly frustrating is the fact that people are able to look at a stick figure performing an animation and appreciate the lifelike motion. However, when that same motion is applied to a synthetic 3D character, those same people get hung up on the look of the character. They no longer regard the motion of the character as realistic simply because it looks "odd." If the character is a monster or something else nonhuman, this problem seems to go away. Well, that's great if you're creating a shooter filled with mutated zombies and uncontrollable robots. However, if you're creating a realistic scene filled with average people, you're in trouble.

This observation has led me to think that for now, at least, the focus for real-time 3D characters should not be on trying to achieve realism. Instead, we should be looking at approaches to creating stylized characters. Perhaps Disney had the right idea. For years, its artists have seemed to understand and appreciate this paradox. They were able to create very realistically painted backgrounds full of color and depth. For the actual characters, though, they still rely on simple pen-and-ink drawings. Even when the first fully CG character was introduced in a Disney animated feature, the magic carpet in *Aladdin*, it was rendered in a style that matched the traditional methods. With this in mind, is it at all surprising that 3D animated series such as Mainframe Entertainment's *ReBoot* and *Beast Wars* focus on robotic and animal characters?

LISTING 1. *A Renderman checkerboard.*

```
//
//        Shader:    Checkerboard Shader
//        Arguments:Diffuse and Ambient Coefficient
//                   Number of squares
//                   Two colors to alternate
//
surface checkerBoard(
   float Kd, Ka;     //  Specular and Ambient Lighting
   float frequency; //  Number of Squares
   color c1, c2      //  Two colors
)
{
   // S and T vary from 0 - 1 across the surface
   float smod = mod(s*frequency,1);   // Interval in S direction
   float tmod = mod(t*frequency,1);   // Interval in T direction

   // Ci is the output color
   if (smod < 0.5) {             // Odd Columns
      if (tmod < 0.5)
         Ci = C1;                // C1 Square
      else
         Ci = C2;                // C2 Square
   } else {                      // Even Columns
   if (tmod < 0.5)
      Ci = C2;                   // C2 Square
   else
      Ci = C1;                   // C1 Square
   }

   Oi = Os;          // Opacity out = opacity in

   // ambient() returns ambient light value
   // diffuse(N, I) returns the sum of lights from
   // incident vector I and surface normal N
   Ci = Oi * Ci * (Ka * ambient() + Kd * diffuse(faceforward(N,I)) );
   // Ci is final color
}
```

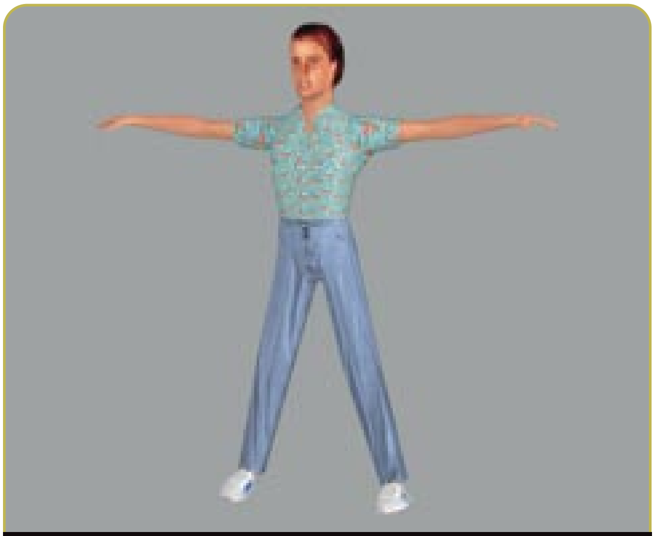


FIGURE 1. *Our original character, outfitted with textures created from photographs and scans.*

Losing a D

Perhaps it is time to look at using 3D technology to create much more stylized animations instead of realistic ones. This technique, called non-photorealistic rendering (NPR) in academic circles, has emerged as a strong research field at industry conferences such as Siggraph. That means there are plenty of fresh, steaming piles of research to get me started.

The character in Figure 1 was created using textures created from photographs and scans. I wanted to get something much more stylized, so we had another model and set of textures created with a cartoon kind of look in mind. You can see the results in Figure 2. This character is much more typical of the kinds of characters you may see in a 3D game. However, it doesn't quite capture the 2D look I had in mind.

For one thing, the shading implies too much depth. The maps really need to be reduced to only a few colors. This is no problem to do in any image processing program as you can see in Figure 3. This is much closer to the idea of a cartoon rendering. However, the image is clearly missing the bold outlines that characterize cartoon images. To create those lines, I need to turn to some technology.

GL to the Rescue

The first lines that I need to create are the silhouette lines. These lines define the outline of the character. On a 3D model, the outline of a model is defined by the model's edges. Intuitively, I know that a silhouette edge must occur when an edge connects a polygon facing forward and a polygon facing backward. This can be expressed mathematically as:

$$(N_1 \bullet (V - E))(N_2 \bullet (V - E)) \le 0$$

where N_i are the two face normals for the adjacent polygons, V is a vertex on the edge, and E is the eye point. When this statement is true, the edge is part of the silhouette.

As you can imagine, this would be a rather time-consuming process on a model that had any significant number of faces, but this method has the benefit of identifying the actual edges that define the silhouette. This could be useful if I wanted to apply some other effects to the silhouette lines. But for now, I want to look for a faster way that makes use of my existing 3D hardware.

I can start by drawing the front-facing polygons with texture. I can then draw the back-facing polygons in line mode. Since the Z-buffer is already filled for front-facing pixels, the only pixels drawn will be those pixels along the edge. However, in order for this to work, I need to set the depth test so it draws pixels that are at the same depth as those in the Z-buffer. In OpenGL, this setting is `glDepthFunc(GL_LEQUAL)`. This gives me a rendering algorithm like this:

1. Draw front-facing textured polygons.
2. Set depth test to `LEQUAL`.
3. Draw back-facing lines.

Or in OpenGL:

```
glPolygonMode(GL_FRONT,GL_FILL);   // Draw Filled Polygons
glDepthFunc(GL_LESS);              // Don´t draw shared edges
glCullFace(GL_BACK);               // Draw front facing polygons only
DrawModel();                       // Call my draw routine
glPolygonMode(GL_BACK,GL_LINE);    // Draw Lines
glDepthFunc(GL_LEQUAL);            // Draw shared edges
glCullFace(GL_FRONT);              // Draw back facing edges only
DrawModel();                       // Call my draw routine
```

You can see the result in Figure 4. The first frame shows just the resulting silhouette lines and the second frame shows the combined image.

With this technique, I can use OpenGL to enhance the effect. I can make the lines thicker or even anti-alias the lines with alpha blending (or even anti-aliased hardware lines, if available). It is even possible to make the lines pop out from the model using `glPolygonOffset()`. However, this

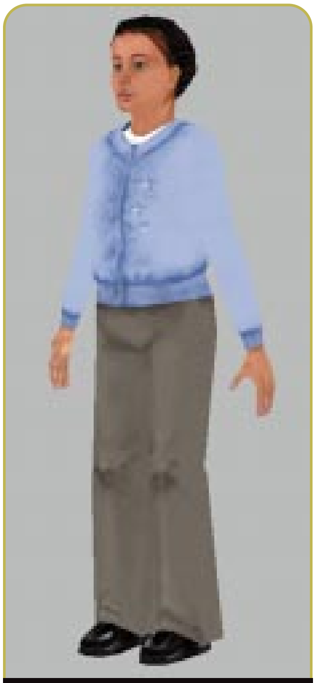


FIGURE 2. *A more stylized, cartoonish version of the character.*

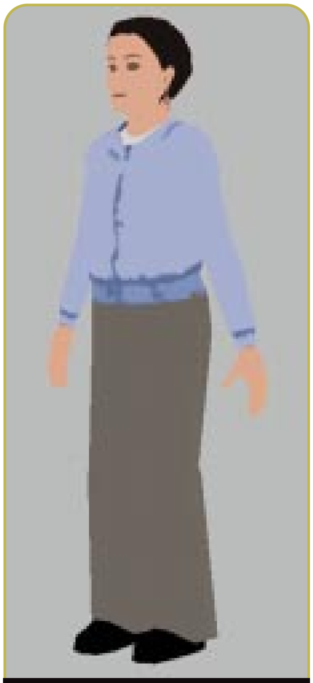


FIGURE 3. *Reducing the depth of colors gives a more 2D look.*

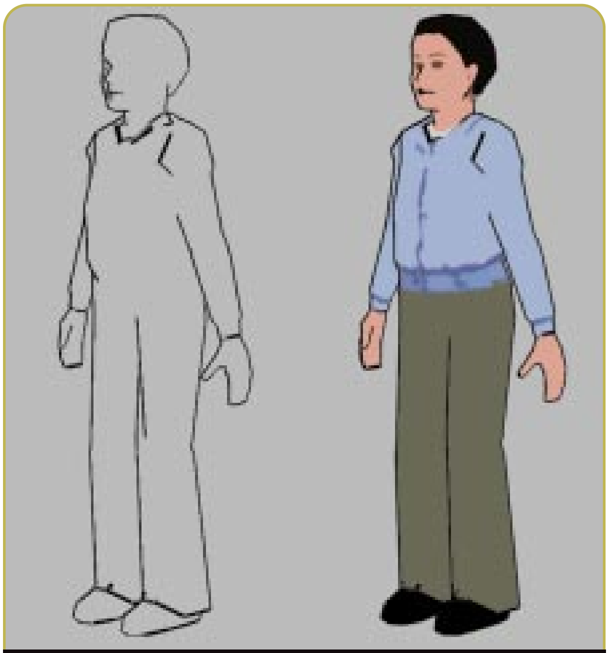


FIGURE 4. *To complete the cartoon look of the character, black outlines were added with OpenGL.*

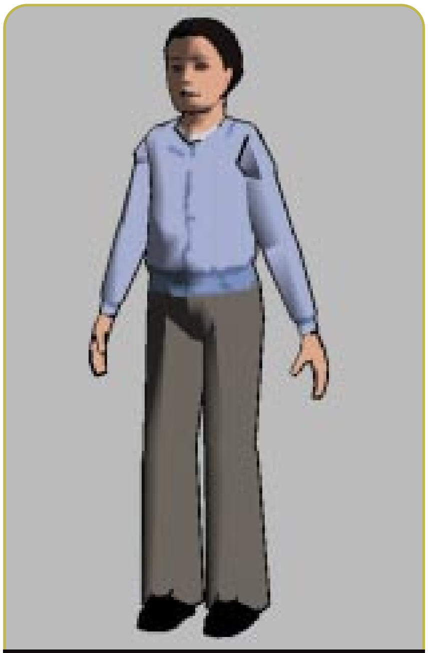


FIGURE 5. *Finally, shading is added with our lighting model.*

can lead to a mess, so you need to be careful.

Another approach that may make cleaner lines requires the use of an extra pass and the stencil buffer. In this technique, the algorithm is:

1. Draw front-facing textured polygons.
2. Set draw mode to stencil only.
3. Draw front-facing edges in line mode.
4. Draw back-facing lines where stencil is set.

This will ensure that only edges that are shared front and back are drawn. However, since the stencil buffer is not commonly available across consumer hardware, it may be wise simply to test for it and use it when possible.

In addition to the silhouette lines, I probably want to add interior lines that define changes in the material of the character. This cannot really be done easily with just rendering tricks. This requires a pass through the object to detect edges that share polygons with different materials. These edges are marked as material boundaries and are drawn after the render. Luckily, the material edges are not viewer-dependent so they can be calculated only once as a preprocess.

Some Shadier Business

Now that I have a nice method for creating cartoon-style characters with silhouette lines, I need to think about shading. Applying typical Gouraud shading to these characters would ruin the effect I am trying to achieve. I need to change the lighting model to make this work.

In the Gouraud shading system, the angles between the viewer, light, and surface normal are used to determine the shade of the vertex. For my simple cartoon rendering, I only want two shades for each material, light and dark. In order to do this, I need to calculate the vertex colors myself. The formula I applied is:

$$\left(N_V \bullet \left(V - E\right)\right) < \varepsilon$$

where N_V is the vertex normal, V is the vertex position, E is the eye point, and ε is the shading threshold. This is very similar to the silhouette-detection formula. However, in this case, when the result of the formula is less than a certain threshold, ε, the vertex is shaded with the "dark" color. Otherwise, the standard color is used. However, since the color interpolates across the surface, this still doesn't look right, as you can see in Figure 5. I want the color to change at a single point across the surface of the polygon scanline. This will require the use of a texture-mapping technique, which I'll get to next month.

The Squashy and Stretchy Show

I think these techniques provide a new way of thinking about real-time 3D animation. It's a classic example of embracing your limitations. There's lots of room for experimentation and exploration. Creating the ideal texture to work with non-photorealistic rendering will require some creativity on the part of artists.

Another intriguing idea would be to apply some soft-body deformation techniques to the models (see "Collision Response: Bouncy, Trouncy, Fun," Graphic Content, March 1999). These new squishy objects can be rendered using NPR techniques to get a real Road Runner feel. For this month, play around with the simple cell shader and begin exploring the world of the less-than-realistic. You can download the source code and the application from the *Game Developer* web site at http://www.gdmag.com. ■

Acknowledgements

Thanks to Lisa Washburn at Vector Graphics (http://www.vectorg.com) and Tom Knight at Imagination Works (http://www.imagination-works.com) for the models and textures used in this article.

FOR FURTHER INFO

Siggraph 1999 had an entire course on non-photorealistic rendering. If you are interested in the topic you should definitely get the notes or the courses CD: "Non-Photorealistic Rendering," Course 17 (Siggraph 1999, ACM Siggraph).

- Markosian, Lee, and others. "Real-Time Non-Photorealistic Rendering," *Proceedings of Siggraph 97*. New York: ACM Siggraph, pp. 415–420.
- Uphill, Steve. *The RenderMan Companion*. New York: Addison Wesley, 1990.

by Mel Guymon

Pyro-Techniques: Playing With Fire

Remember that time your mom told you not to play with matches, but then you and your friends got together and accidentally burned down the neighbor's doghouse? In this month's column, we're going to build an even bigger fire, and we promise your mom won't ever find out.

Where There's Smoke...

First off, we need to identify exactly what kind of fire effect we want to generate. For our example, we'll be generating a bonfire in a real-time 3D environment that has a photorealistic art direction. Therefore, our fire effect will need to look as true-to-life as possible. Second, we'll sit down with the programmers and outline what tools and parameters we'll have to work with. In this example, we'll have access to and take advantage of the following: 32-bit texture maps up to 256×256 pixels, animatable sprites, RGB vertex color, vertex alpha, additive blending, and dynamic projection lighting. Last, we need to work with the programmers to generate an in-game particle system capable of pulling off the effect. In this case, we've prototyped the effect in an off-the-shelf product, 3D Studio Max, and the programming team has duplicated the required functionality demonstrated within Max's particle system.

Now we're ready to begin. There will be three main entities for our bonfire: flames, smoke, and sparks. Each will be created with groups of quad polygons, on which an animating sprite sequence has been mapped (in the case of the fire, this sequence will need to loop). The flames and smoke will be generated by clustering and overlapping these polygons in a random and chaotic manner. Why do we choose this method? Why not simply make a large polygon for the smoke and another for the fire? Because suspension of belief must be maintained as far as possible. Unless we have an inordinately long sequence of sprites, the viewer is easily going to be able to discern the looping pattern. Also, without some kind of random perturbation to the effect, every fire will look identical — boring. In Figure 1, you can see the effect we've outlined. On the left are the untextured polygons, with yellow representing the flames and blue representing the smoke. Notice that in the in-game shot on the right, it's difficult to identify the boundaries of a single individual polygon. This is possible because of how we've created our texture maps, which is the next step in the process.

Once the technique for creating the effect has been determined and the programmers have created the toolset, it is up to us as artists to build and iterate on the effect to get it looking just right. Before we start creating the textures, though, it's good to have some reference material from which to work. Fortunately, there are many graphical references for flames and explosions, as free downloads or from within pregenerated

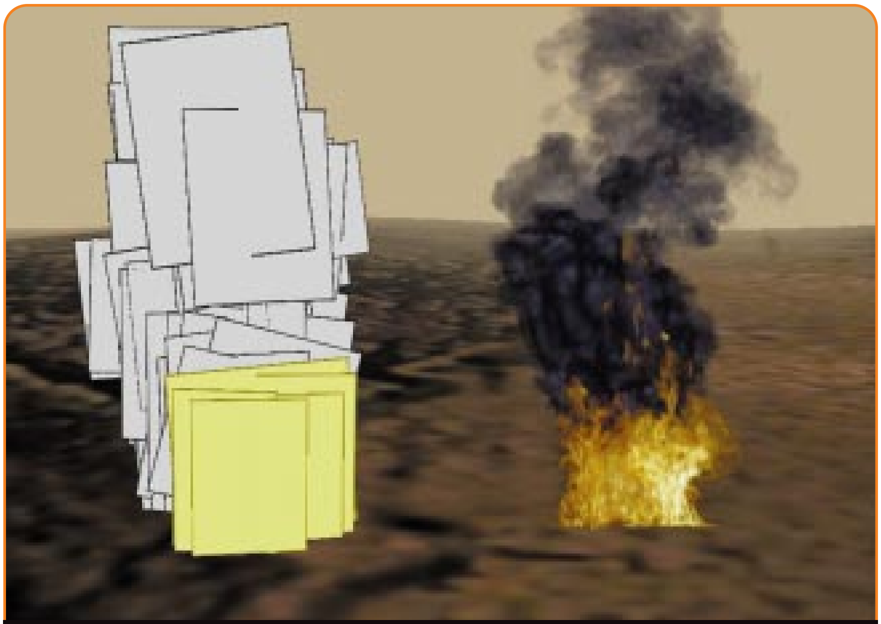


FIGURE 1. *Behind the scenes of the flame effect: the yellow polygons represent the flame, the blue ones represent the smoke.*

Mel has worked in the games industry for several years, and recently finished work as the art lead on DRAKAN. Currently, he manages a modeling and animation studio which provides custom content for RT3D games. Contact him at mel@infinexus.com.

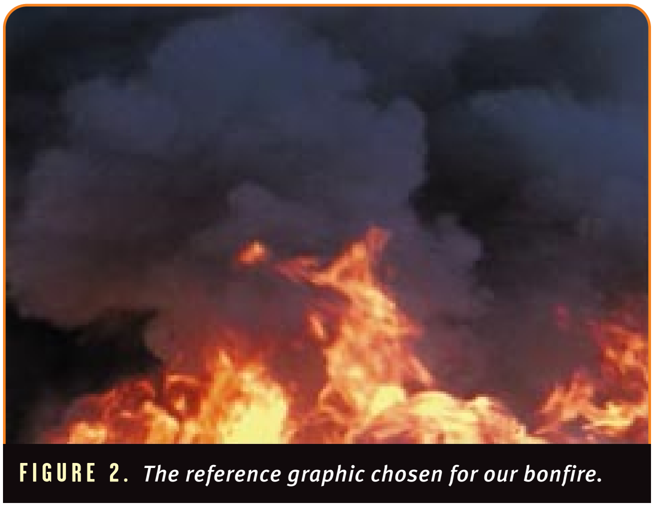


FIGURE 2. *The reference graphic chosen for our bonfire.*

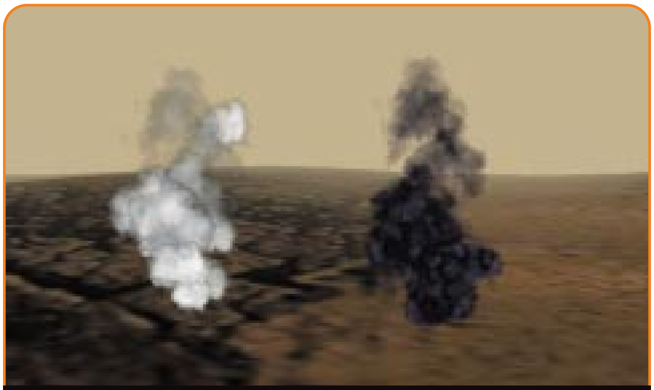


FIGURE 3. *A combination of a procedural texture and an alpha map make an effective smoke particle system (left). The in-game effect is shown at right.*

sets (Pyromania, ReelFire, and others). For our example, we'll be using texture reference from an off-the-shelf product, Pyromania 2, while the inspiration for the effect will come from a picture that was downloaded for free (Figure 2).

For the smoke entity of our effect, we'll use a nonlooping sequence of animated sprites 30 frames in length. (The length is arbitrary and should be tweaked to optimize the visual effect within the texture-memory requirements for the engine.) The smoke's color will be matched as closely as possible to that of the reference graphic, a dark bluish color. We will approximate the voluminous, billowing appearance of the smoke by the patterns in the texture, which were generated using a prerendered noise material in 3D Studio Max. Smoke has a fractal appearance, which means its edges appear soft and undefined — as you zoom into look at the edge, you keep finding more and more detail. To approximate this, all of the alpha maps used for the smoke will have soft, fuzzy edges. (In our particular case, the smoke's alpha components have been derived from a Pyromania 2 smoke effect.)

In Figure 3, you can see how the combination of using a procedural texture with a canned alpha map hides the individual texture boundaries, making the smoke particle system effective. On the right is the smoke effect as it will appear in the game. On the left you can see the results of using a texture with its originally-associated alpha map. Though the rendered effect for each individual texture is perfect, the textures have easily discernable boundaries, and when composited together resemble nothing if not a cluster of cotton balls. The smoke polygons are generated from random locations within the volume of the smoke emitter (this can be an arbitrary point in space, or can be defined by an artist as a geometric object). The speed at which the smoke animation is played, and each smoke polygon's vertical and horizontal speed, scale, and rotation, are all slightly randomized to achieve a less uniform look. When the textures on each polygon reach the end of their animated sequence, the polygon vanishes.

For the flames entity of our effect, we'll use a looping sequence of sprites 12 frames in length (as with the smoke, the sequence length is arbitrary). We'll match the flames' color with our reference graphic and the shape of the flames will be derived from a sequence of fire animations from Pyromania 2. It's important to note here that in the case of the smoke textures, we had several options available for generating the smoke textures and alpha maps. Smoke has a generally random, fractal pattern, and there are many programs that can generate realistic-looking cloud or smoke-like effects. In the worst case, I could have painted the smoke textures by hand in Photoshop or some other paint program with little or no degradation of the effect. For the fire, however, we're better off using an off-the-shelf, canned effect, at least as a starting point. The random, fractal nature of fire is extremely hard to duplicate by hand, and although current procedural methods for gener-

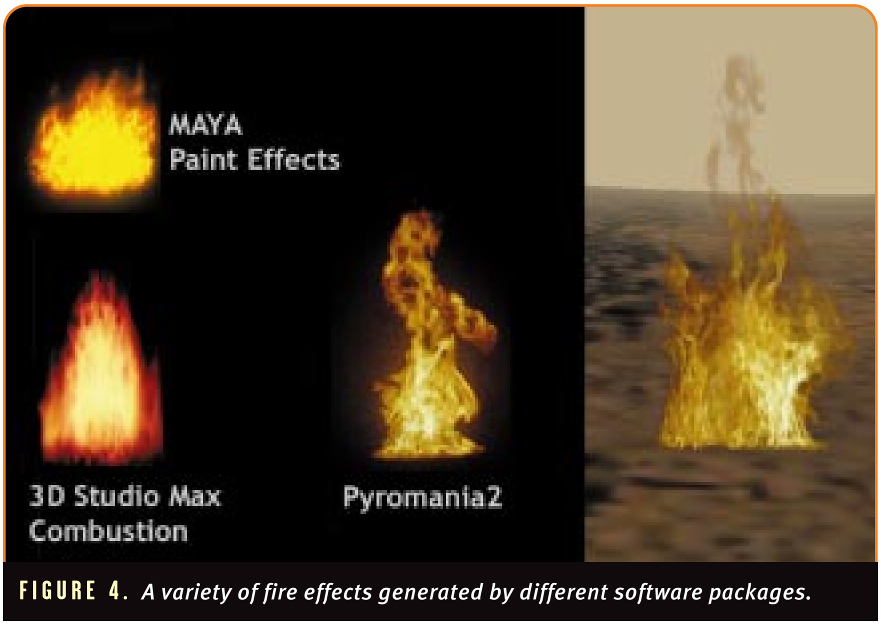
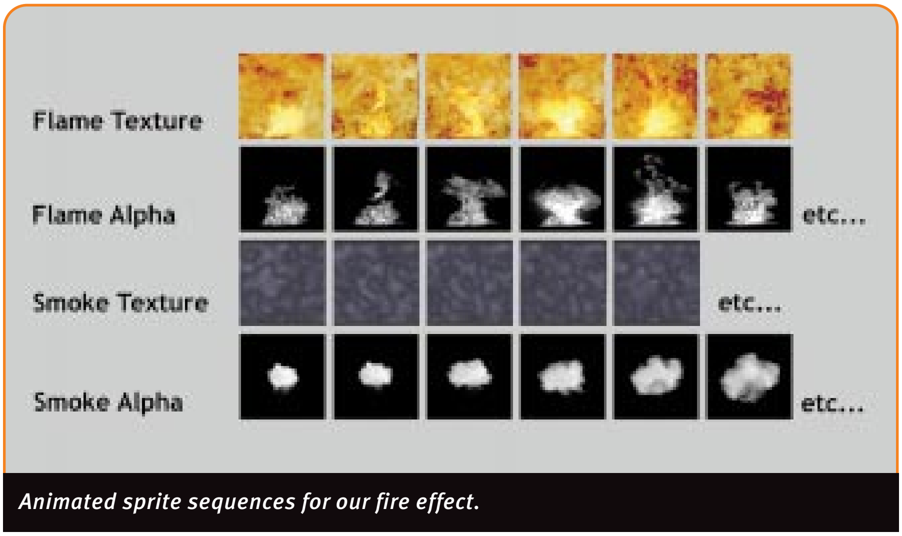


FIGURE 4. *A variety of fire effects generated by different software packages.*



Animated sprite sequences for our fire effect.

ating flames do a fair job, the effect falls far short of the real thing, as you can see in Figure 4. The first two groups of flames were generated by the standard flames particle systems in Paint Effects (Maya 2.5) and Combustion (3D Studio Max 3). As in the case of the smoke effect, the flame polygons will be generated with a random scaling factor, so that their horizontal and vertical size will vary slightly, although the lower edge of each polygon will be aligned. The flame polygons will also be different from the smoke polygons in that they will remain rooted to the ground throughout the entire effect. The rising nature of the flames, in this case, has been represented within the texture and alpha maps.

Adding the sparks entity to the effect will provide an added degree of randomness and break up the scale of the polygons involved (the polygons for the flames and smoke are about the same size to within a single order of magnitude). There is no need to generate additional textures for the sparks; textures from the flames and the alpha maps from the smoke can be combined for an excellent effect, and the colors will automatically match. The spark polygons are generated in much the same way as those for the smoke, though the scale is much smaller and the velocity much higher. With the textures created and in place, we've got a pretty decent bonfire going. Now it's time to use the remaining non-texture-based tools in our toolbox: vertex colors, vertex alpha, additive blending, and dynamic lighting.

Vertex Colors and Vertex Alpha

The vertices on each polygon can be colored to add graphic variation without additional texture expense. Most mainstream production tools support vertex-color editing. In our case, however, the vertex colors will work just as well. In any case, the vertex colors for our fire will be adjusted procedurally. I've found that this technique works best with the smoke polygons. Smoke is actually composed of fine particles that receive light and cast shadows. Consequently, if the smoke is thick enough, its color will actually be affected by light cast by the fire — the smoke will be lit from beneath as it rises. To achieve this, the vertices at the bottom of each smoke polygon will be varied as a function of their height from the fire; the closer to the fire, the more drastic the effect. In our case, the lowest vertices of each smoke polygon will be colored reddish-orange in accordance with the color of light being cast by the fire. In Figure 5 you see the effect. Note that to make full use of this effect, the smoke polygons have been subdivided. This is a level-of-detail option that can be turned off as the player gets farther away from the effect. Depending on the engine and implementation, you may need to lighten the smoke textures to see the effect of vertex colors. Although we've applied this effect only to the smoke polygons, the same procedure can be used to enhance or alter the colors in the flame and sparks polygons as well.

With the random generation of particles, there is always the problem that a smoke polygon will be created which goes off on its own, too far from its siblings. If this happens, the viewer may easily discern the polygonal nature of the effect and the suspension of disbelief will be lost. To minimize the impact of this, the errant polygon can be forced to fade out early. This can be done on a per-object level, but a more seamless effect can be achieved by sequentially fading out the individual vertices of the polygon as it crosses a defined threshold.

Additive Blending

Fires cast light, which is why we can see them even from a great distance. And though we can't effectively replicate this within the confines of a pixel-based 3D environment, we can certainly fake it. In order to make the flames appear to cast light or glow, the flame pixels must be arbitrarily brighter than the pixels surrounding them. For us,

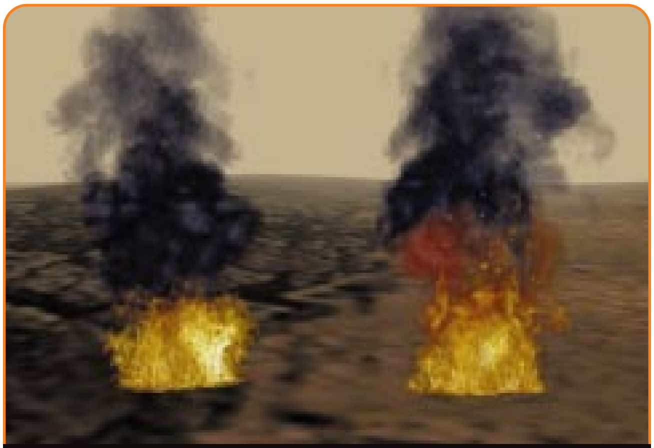


FIGURE 5. *Adding vertex colors to the smoke polygons (shown at right) gives the smoke the appearance of being lit by the fire beneath it.*

this is equivalent to saying the flame pixels must be arbitrarily more white. In fact, the brightest we can ever make an object appear is to force all its pixels to read RGB = 255,255,255.

Additive blending can achieve this for us. When pixels are assigned to have additive blending, they increase the brightness, or whiteness in this case, of the pixels onto which they are blended. Thus, by compositing these pixels in several layers, each succeeding layer creates a brighter, whiter effect.

This is a perfect application for our flame polygons. Since they are clustered and overlapping, the densest portion of the flames will have the brightest (most white) pixels. This is exactly what we want to achieve, since it will help us mimic the light-casting nature of real-life flames. Figure 6 shows the result of additive blending in contrast with the previous version of the flame effect. (This effect can even be further improved by adding a slight halo or lens flare to the flames.)

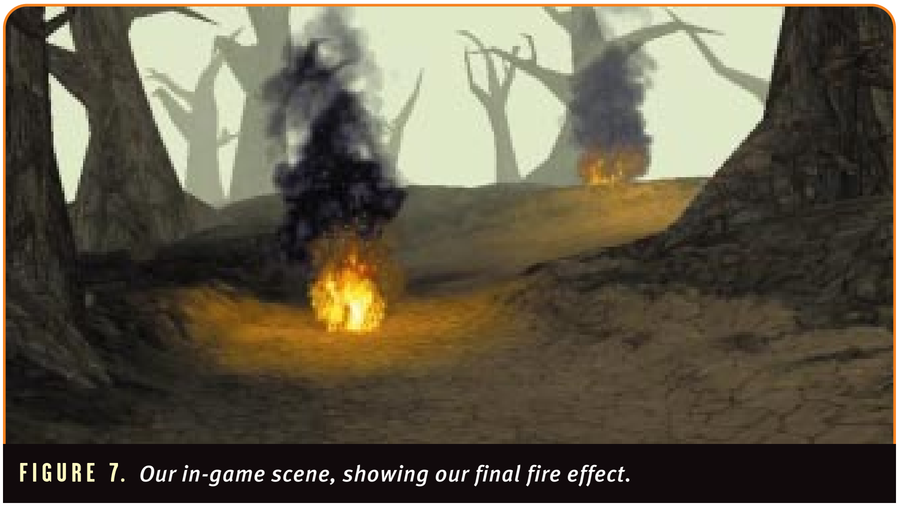


FIGURE 7. *Our in-game scene, showing our final fire effect.*

Dynamic Projection Lighting

The light cast from a fire jumps and dances in concert with the chaotic motion of the flames. In order to approximate this effectively in the game, we need to use a projection light. The actual implementation of this effect can vary widely with each engine, but in essence the projection light should derive its pattern from the same sequence of alpha maps as that of the flame polygons.

When this is done correctly, the flames will "dance" on any surface within the range of the light. The effect of the dancing lights, although difficult to represent in a static image, can be stunning in game, and will greatly enhance the realism of the effect. The final in-game product of our fire effect, incorporating all of the above techniques, is shown in Figure 7.

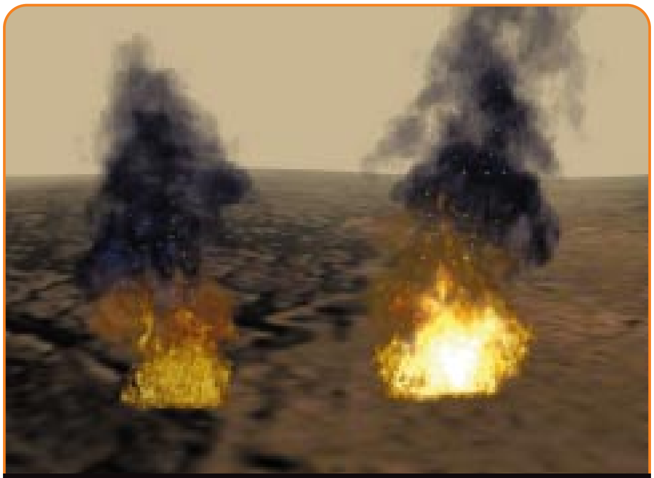


FIGURE 6. *Additive blending techniques have been added to the flame pixels, creating the effect seen at right.*

Fanning the Flames

Many of the effects we used in this month's example were not possible a few years ago. The technology simply wasn't there, or didn't have enough of a user base to be applicable on the target platform. However, the pace of technology shows no sign of slowing down and we are always being presented with more tools and methods for content generation. We need to ask ourselves constantly "What if?" so we'll be better prepared to apply new technologies and become more creative and efficient developers. ■

Electronic Arts: Infinite Channels

There's an old Gary Larson *Far Side* cartoon that shows two proud parents watching over their child who's playing some console game, and the parents are imagining want ads offering lucrative careers to anyone with videogame experience. The notion that long-term videogame enthusiasm could result in eventual career advancement seemed pretty ludicrous. That was a long time ago.

In November 1999, CMGI, an investment holding company with a range of Internet interests, gave $11 million to Dennis Fong, the champion QUAKE player better known as Thresh, to start Gamers.com, a game portal. Around the same time, Lycos paid more than $200 million for another online game portal, Gamesville. In the space of just a few weeks, the online gaming market became the next hot Internet investment. Online gaming is now an important part of the Internet market, and no more so than in the impact it may have on a company that plans to get 20 percent of its revenues from cyberspace, Electronic Arts.

Company History

EA was founded by Trip Hawkins in 1982 and funded by legendary Silicon Valley venture capitalist Don Valentine to the tune of $2 million. It took four years for EA to become the United States' largest publisher and developer of PC games. In 1989, EA got on board the Sega Genesis with JOHN MADDEN FOOTBALL. EA's sports titles were critical to the success of the Genesis and helped propel the company into the more lucrative markets of console games. In 1990, EA also began developing for Nintendo, so that by the end of 1994 three-quarters of EA's more than $400 million in revenues was coming from console games.

Today EA is the world's leading independent game publishing company. The company markets its products under seven brand names: EA, EA Sports, Maxis, Origin, Bullfrog, Westwood Studios, and Jane's Combat Simulations. It distributes products in 75 countries and has development studios and partners in North America, Europe, and Asia. EA is the company against which all other U.S. publishers are measured. Now, it appears, EA is going to lead the charge in online gaming. It's been a while coming, but if EA is ready to do business on the web, now's as good a time as any.

Online Gaming Heat

In November, EA signed an agreement with America Online to become the premier online game provider for the country's largest ISP. This news came just after the company had agreed to take over Kesmai, a leading developer and provider of multiplayer online games. Of course, EA has never made any secret of its online business ambitions, and it is the only major game publisher to have provided a business model that the game industry and the financial community can respect. That workable business model owes a lot to ULTIMA ONLINE, which continues to deliver subscriber revenues to EA while many other online game initiatives falter. Mpath.com, which I analyzed some months back (Hard Targets, July 1999), is now more of an online audio company than a game company. Total Entertainment Network recently became Pogo.com and has shifted its focus to the casual-gamer market. The online world is littered with businesses trying to take advantage of the promise of interactive entertainment.

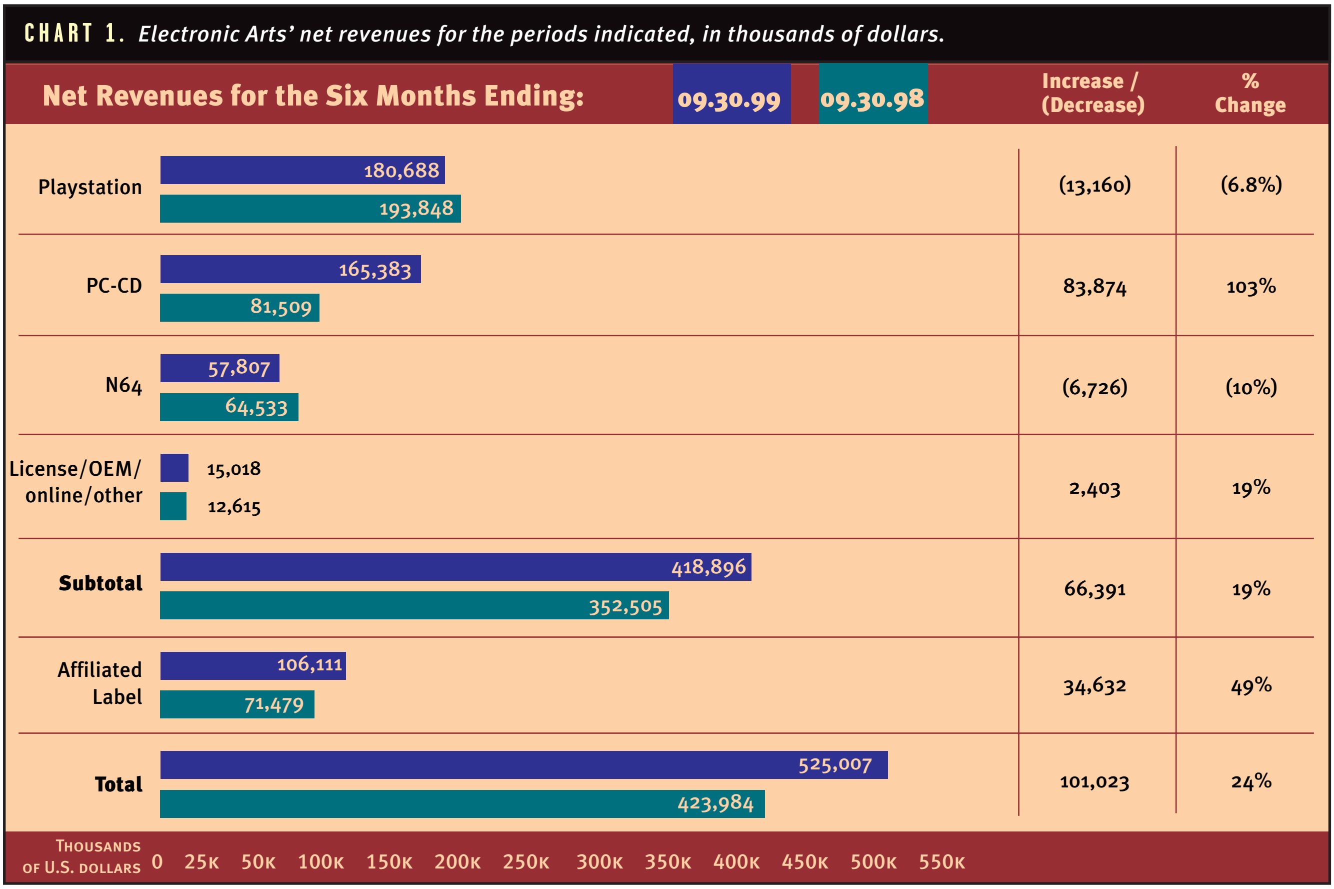
The secret to EA's strength may be quite simple: the company is, without doubt, the most recognized game publisher in the world after the big three console makers, Sony, Nintendo, and Sega. Although most of EA's products are targeted at the traditional game enthusiast, it has a blue-chip reputation with casual game players, too. When EA made the deal with AOL, EA's stock took a big leap. All the equity that is invested in the EA brand can easily translate into a magnet for online crowds.

At the end of the day, the greatest competitive advantage that EA has is that it has shown consistent strength across platforms and content genres. In the highly diverse macrocosm of online gaming, few companies have the same depth of experience. Most importantly, EA has excellent distribution channels. EA's distribution strength will be what ultimately assures its success in the online world, particularly if you consider the

Omid Rahmat is the proprietor of Doodah Marketing, a digital media consulting firm. He also publishes research and market analysis notes on his web site at http://www.smokezine.com. He can be reached via e-mail at omid@compuserve.com.

by Omid Rahmat

CHART 1. *Electronic Arts' net revenues for the periods indicated, in thousands of dollars.*



Internet as one of many channels to the consumer.

Distribution Magic

Unlike most of its competitors, EA distributes all of its own products. The company has a vast distribution network covering nearly a hundred territories on six continents. In addition, EA serves as a distribution partner for many other game publishers through its affiliated label program as well as through co-publishing deals. In most of these cases EA holds the upper hand. Strength in distribution is a key characteristic of any manufacturer that wants to be a success on the web, and EA has plenty of it.

For instance, EA offers direct distribution services in North America whereby a retailer has EA deliver directly to the retailer's outlets as opposed to the retailer's distribution centers. This arrangement saves the retailer significant costs associated with handling and shipping products and ensures rapid delivery to satisfy consumer demand. It also gives EA tremendous experience in addressing consumer demand directly. The company uses two large distribution centers in the United States: one in Hayward, Calif., and one in Louisville, Ky. These centers handle all order processing for worldwide distribution. They receive finished goods from the contracted duplication and packaging vendors and warehouse the products until they are needed to fill a specific order. They handle EA-branded products, co-published products, and affiliated label products. Advanced information systems, workflow processes, and proprietary delivery scheduling software provide distribution-center managers with powerful and efficient tools for performing their activities. EA contracts with all major overnight and traditional delivery services, such as UPS and Federal Express, for the physical delivery of goods from its distribution centers to the customers' locations, minimizing investment in mobile assets while continuing to deliver industry-leading distribution activities.

At present, no competitors have been able to match the entire collection of activities associated with EA's continued production of hit products. The company's combined use of Electronic Data Interchange (EDI) and direct store delivery allows retailers to place online orders any time of day and receive those orders directly from EA faster than they would from any other publisher. While competitors have the ability to utilize EDI and direct store delivery, none has matched the scope of EA's distribution program.

In effect, EA has already figured out how to handle the demands of a global market and a variety of direct channels to the consumer. Of course, online gaming isn't just a matter of distributing content. I just want to make the point that the logistics of distribution are primarily the same as the logistics of handling any direct interface with customers, be it Electronics Boutique or John Q. Public.

Supporting the World

The real cost of online gaming is in supporting the consumer. In making a pact with AOL, EA has basically given itself a buffer to the widest possible audience on the web today, the AOL subscriber base. AOL is EA's shop window on the Internet. This removes one level of one-on-one support that EA doesn't have to worry about: managing relationship marketing. By that I mean AOL is the first point of contact. The fact that both EA and AOL are blue-chip companies in their respective markets makes an alliance between the two very promising. However, EA has gone on to do something even more interesting, signing a licensing agreement with Marimba, a provider of Internet-based software management tools. This agreement allows EA to use Marimba's Castanet technology for integration with its growing library of online content.

Initially, Castanet technology will be used to create a seamless patching system that automatically updates software to ensure that EA's customers are using the most up-to-date version. As time goes by, EA is going to leverage Castanet to manage the integrity of its game software at launch. It's a very compelling message, and the evolution of EA's online strategy implies that the company has taken full advantage of its expertise in the distribution and support of packaged goods to create a set of criteria to meet the demands of building customer relationships online.

It should also be noted that with Ultima Online, EA has suffered a degree of criticism on issues related to the quality of software and the level of support it has provided online users. Some users went as far as to take EA to court.

The top ten game publishers in the world today have all evolved to emulate Electronic Arts in some ways.

Whatever EA's past problems, no one can accuse the company of not learning from its past mistakes. It has experience with paying online-gaming customers, a world-renowned consumer brand, and successful franchises in almost every genre and on every platform. Online gaming can easily become one more avenue of distribution, and one more interface to EA's consumers.

Perhaps the stock market's enthusiastic reaction to EA's deal with AOL was exaggerated. The company isn't doing anything revolutionary. In evolving from its existing distribution and support structure to one that incorporates a one-on-one interface to consumers — through AOL and the Internet — EA is setting down a blueprint for every other game publisher, much as it has done in building its present global infrastructure. The top ten game publishers in the world today have all evolved to emulate EA in some ways. They may not all agree, but you can't argue with success. In the near future, online gaming may also evolve to emulate the EA way. After all, isn't Asheron's Call, or Everquest for that matter, the heir to Ultima Online? ■

Back

Physically Modeled Audio

by Mark Miller

While much ado has been made over the past few years about physically modeling the interactions between 3D objects in games, not as much attention has been paid to another form of physical modeling — that of sounds. And according to some people in the game development industry, physical modeling synthesis is the "next big thing" in interactive audio. In the synthesis world, wavetable produced a huge sound-quality increase over Frequency Modulation (FM) audio. More recently, downloadable sound (DLS) technology broke open the sonic boundaries of General MIDI (GM) wavetable synthesizers. Yet even wavetable-plus-DLS solutions do a fairly questionable job when it comes to rendering certain critical instruments and sound effects in both a realistic and responsive fashion. Red Book audio may be quite good at capturing individual performances of such sounds, but it is still too large for many applications and too difficult to manipulate for interactivity in all but the simplest ways.

Mark Steven Miller has been studying, producing and commenting on interactive media since 1989. Having participated in the development of more than 100 cutting-edge titles, Mark is currently the president of Group Process Consulting which provides technical and strategic consulting services to new media companies. More about his current project, the Interactive Media Information eXchange (iMIX), a developers' association for producers of interactive television content, can be found at http://www.imix-tv.org. Mark can be reached at mark@GroupProcess.com.

Fortunately, some recent advancements in physical modeling synthesis technology and new products that incorporate this technology may provide the answer.

Physically modeled audio offers a solution that other audio technologies cannot match. Typical FM synthesis is too abstract and limited to portray complex, real-world sounds accurately. Wavetable and its big brother, Red Book audio, are quite good at such portrayals but both suffer from being overly literal — analogous to a photograph for visual representation. Photographs are excellent for representing a particular image as seen from a particular angle, under a particular set of circumstances. However, unlike a scene created in, say, 3D Studio Max, photos do not contain the abstract, elemental forms or structures that make up the image. As a result, the data in a photograph is fairly useless if you want to know what the subject would look like seen from the other side or under different lighting conditions.

Physical modeling, on the other hand, is much more like real-time 3D graphics in that sounds are generated based upon complex mathematical models of the way sound is created in the real (or surreal) world. The "modeling" aspect implies that the sounds are generated based upon abstract constructs of the objects and/or processes that create the sounds. The "physical" part implies that the object or process exists as, or can be extrapolated from, real-world (or surreal-world) objects or processes and will obey the applicable laws of physics when so stimulated. A good example of this would be a physically modeled four-cylinder car engine sound effect that "revs" correctly when you "step on the gas." The combined result is a flexible method for generating accurate and complex real-world sounds that respond in sensible ways to appropriate stimuli.

How could physically modeled audio benefit a game? Imagine hyper-realistic race car engines that rev smoothly and automatically, and accurately modify themselves as the car takes damage. Or stadium crowds that respond exactly how you think they should to simple "excitement" and "tension" sliders. Or a dead-on perfect electric guitar that reproduces the feedback from Jimi Hendrix's "Star Spangled Banner" interactively, just as a knight brandishes a magical blade.

This article looks at two companies, Yamaha and Staccato Systems, which offer physical modeling software synthesis products to the game development community, and at some of the game companies using these and other advanced synthesis technologies.

Staccato Systems' SynthCore

Staccato is a small start-up company with roots in the Sondius project, which itself was born out of Stanford University's Center for Computer Research in Music and Acoustics. The Sondius program was initiated in 1993 to develop physical modeling technology, algorithms, and development tools. In 1996, the Sondius team left Stanford and incorporated as Staccato Systems with a license to 20 fundamental patents, the Sondius Logo, and source code for some of the tools that they had developed. In 1997, Yamaha entered the picture and created the Sondius-XG partnership with Stanford. In turn, Staccato upgraded its license to include the new Yamaha intellectual property (which includes more than 400 patents and the XG-Lite sound set). Around the same time, Staccato also received a first round of funding from Yamaha of Japan. Lately, the company has accumulated engineering and marketing talent at an astonishing rate. Former staffers from E-mu, S3, Yamaha, Apple, and other major music technology companies have recently joined Staccato.

Staccato's initial product, the SynthCore SDK, draws on the last 25 years of research in the broad field of algorithmic synthesis. It features physical modeling, modal synthesis (primarily for percussion sounds), waveguide synthesis (primarily for plucked strings), physical process and event modeling, virtual analog synthesis, and traditional wavetable synthesis capabilities. Do not, however, be fooled by these academic roots. Two of Staccato's primary goals in developing the SDK have been to optimize their code for speed and efficiency, and to provide appropriate and streamlined interfaces (APIs) to and from the game application. The SynthCore SDK is priced at $195 and subsequent use of the technology is licensed on a standard per-title/per-platform basis. Custom sound/algorithm development pricing is negotiable and is based on the time and complexity of the sounds needed.

SynthCore can play back industry-standard downloadable sounds (DLS) and also provides the technology to play back physically modeled sound, which Staccato refers to as downloadable algorithms, or DLAs. In the future, downloadable effects (DLEs) will also be supported. DLEs are programmable audio Digital Signal Processing (DSP) effects such as custom reverbs or filters. For DLS playback, an efficient wavetable synthesizer engine is provided. The SynthCore API can load DLS banks and then control them with MIDI note and controller data. The DLAs, however, are where the real action is. DLAs are script files which describe the configuration of Staccato's flexible Algorithmic Synthesis Engine. In other words, a DLA is a analogous to a patch description for Staccato's physical modeling playback engine. DLAs can also encapsulate DLS files in order to provide an enhanced control structure for more varied effects.

DLAs are designed in Staccato's GUI-based tool, SynthBuilder. Due to the complexity involved with designing a physical model, SynthBuilder is not currently licensed to customers; the Staccato team uses SynthBuilder to design the core models both as an ongoing library development effort and as a service to their customers. In addition, Staccato provides a preset-editor called Mission Control. Mission Control loads DLAs and provides a visual control panel for adjusting any editable parameters. This allows a sound designer to tweak the parameters and to audition the model either with MIDI data or directly from the interface.

Most often, these editable parameters are high-level controllers that work in combination to scale numerous low-level controllers. This allows the designer a reasonably wide degree of freedom with which to exercise the useful modifications that can be made to any particular model without breaking it (Figure 1). The edited results can then be resaved as new presets. Note that due to the extreme mathematical complexity — and hence power — of physical models, most of the unconstrained edits one might make to any given model would render the output entirely unusable. So, while not being a full-blown editor, Mission Control is more like an editor/librarian program that a sound designer would use with an outboard synthesizer.

Once designed, the DLA and its presets can be handed off to the game programmer to be loaded at run time and controlled by the game code. According to Staccato's senior software engineer, Danny Petkevich, "The beauty of the SynthCore API is that it is very thin; you can allocate, assign, and then control via the get controller and set controller methods. All the algorithm's control is encapsulated in the algorithm itself. Sound designers only need to use controllers that make sense (RPM, load factor, engine resonance, and so on) as opposed to terse API calls." On the PC, DLS and DLA outputs are combined and sent either to the fixed or downloaded DSP effects and/or to a DirectSound secondary buffer.

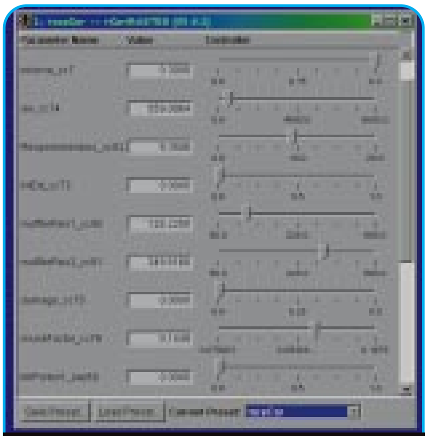


FIGURE 1. *Mission Control race car DLA edit window, showing the exposed sliders of the race car physical model.*

While SynthCore is currently able to handle both music and sound effects equally well, Staccato has chosen to focus on hard-to-solve sound design problems first. First among these are game-critical sound effects that are complex, continuously variable, and require constant control by the game engine, such as race car engines. This type of sound effect has traditionally been very difficult to create using either wavetable-plus-DLS or, gulp, FM. Because such sounds need to be incredibly responsive to the player, Red Book and digital audio streaming have also been out of the question. The beta version of the SynthCore SDK ships with a variety of such models including race car engine, sports car engine, single-piston engine, prop plane, war plane, jet plane, helicopter, rumble, brook, rain, sea, wind, door, and analog synthesizer. It's also worth noting that each of these models can generate a wide variety of presets with Mission Control.

Electronic Arts was the first company to license SynthCore's car engine models for use in its NASCAR REVOLUTION titles. Rob Hubbard, EA's audio technical director, said the decision to license SynthCore was based on "pushing the leading edge in terms of delivering a better and more entertaining experience to the end user. The Staccato technology allows us to push the audio quality up several notches, especially for things like complex engine sound effects. We believe that modeling technologies make it easier for game programmers to hook up the audio and they also allow sound designers more control in the creation and implementation." As an example of CPU load for the models, the race car engine model that EA used required roughly three percent of a 266MHz Pentium II.

The SDK does not ship with many finished musical instrument models, which is unfortunate because the ones that I have heard are quite remarkable. The one exception to this is the virtual analog synthesizer model that ships with the SDK. According to Staccato, this is intended primarily for sound-effects playback rather than for composing music. The reason cited for not pursuing this direction for now is that SynthCore is currently a software-only SDK. As such, it has a certain degree of latency, typically fixed at 70 milliseconds. Controlling SynthCore with real-time MIDI data exacerbates the latency problem. As a result, it is not practical to play or compose music in real time with Mission Control as one would play or compose with a hardware-based synthesizer because the lag time between "hitting a key" and hearing a note is too long for comfort.

Staccato developed a technology called event modeling to enhance wavetable-based sound effects. In this case, an event is a complex sequence of sounds that occurs over time, such as the roar of an audience or a car crashing into a wall. Rather than trying to physically model each of the implied sound generators (such as different hand claps, whistles, and cheers for an audience roar) which would severely tax a typical CPU, the higher-level event is reconstructed out of small DLS components. The event model maintains audio control via a rule set (or algorithm) which specifies the volume, pitch, pan, and time placement of the selected component sounds during playback.

The event modeling algorithm can also generate automated responses to relevant stimuli. For example, an event model of a crowd at a football game can respond in a complex, appropriate, and real-time manner to simple input from the game program, depending on user-defined parameters such as a "quiet/riot" axis or "tension level" axis. While this technique is not new (sound designers have been reconstructing complex sonic events using MIDI files and component samples for some time), the added value that event modeling brings to the task is the ability to adjust the playback of component sounds simply and dynamically in real time, according to sensible behavioral rules.

The PC version of SquareSoft's FINAL FANTASY VIII will include Yamaha's S-YXG50 Plus VL engine.

In addition to its ongoing development of new models for library and custom applications for the PC, Staccato is currently porting SynthCore to the Playstation 2, and the company plans to bring its technology to the Dolphin, Linux, and Macintosh platforms and has indicated that a Dreamcast version might be developed as well.

Yamaha's Virtual Acoustic Technology

Yamaha distributes both hardware and software products that utilize physical modeling (or in Yamaha parlance, Virtual Acoustic (VA) technology). In hardware, Yamaha offers the VL-70m which is a one-half rack-space monophonic physical modeling synthesizer module intended for the professional market. The VL-70m is also compatible with the company's software wavetable synthesizers, the S-YXG50 Plus VL (featuring a single note of VA) and the S-YXG Poly-VL (featuring eight notes of VA).

The S-YXG50VL feature set goes well beyond the current DLS Level 2 (DLS 2) standard. In addition to wavetable DLS playback and physical modeling, the S-YXG50 series adds the extended XG architecture for DSP effects, voice presets, and real-time expression controller mapping. According to Mike D'Amore, Yamaha's business development manager for multimedia products, "physical modeling expands upon the expressive capabilities of standard wavetable playback by allowing for the physical input provided by the musician during a performance to be brought into the electronic world and expressed realistically by the modeled instrument."

The low cost of the S-YXG50 allowed Yamaha to obtain a fairly good OEM share for new PCs entering the market. D'Amore says that the S-YXG50 could ship with 15 percent of all new PCs by the fourth quarter of 2000. While these numbers indicate good progress for OEM distribution, they probably will not represent the kind of platform ubiquity required by most game companies to develop for a nonstandard product. To accommodate this, Yamaha can provide developers with either a fixed-fee or per-title license to distribute the S-YXG synthesizer with their games. SquareSoft of Japan, for example, is licensing Yamaha's S-YXG50 Plus VL engine and will include it in the upcoming PC version of FINAL FANTASY VIII.

While Yamaha expects its hardware products to be used in traditional recording and MIDI studios, the software-based products are intended strictly for run-time rendering of MIDI files on the target machine. One reason for this distinction has to do with the most commonly cited pitfall of software synthesis: latency. Mike D'Amore explains that the current degree of latency (35 to 80ms) remains because of pricing concerns. To get the price of the S-YXG synthesizers low enough for OEMs, certain optimizations simply cannot be made. One can see this cost/performance distinction clearly when looking at more professionally-oriented software synthesizers such as Seer Systems' Reality and Steinberg's Rebirth, which run around $500 each. In addition to being much more flexible and feature rich than either Staccato's or Yamaha's offerings, they feature extremely low latencies which make them, essentially, real-time devices. Fortunately, the S-YXG software synthesizers and the VL-70m are

Harmonix Music Systems' FREQ

Picture the omnipresent airship video screens that hovered above the dying city in the movie *Blade Runner*. Then add a visualization of a high-speed trek through *Neuromancer*'s Cyberspace and, finally, mix in a pumping, interactive techno soundtrack. The result describes my first impression of Harmonix Music Systems' new music/action product, FREQ. I caught up with Greg LoPiccolo, Harmonix's vice president of product development, to talk about FREQ and its use OF physically modeled audio.

MARK MILLER: It seems like this game will be very demanding on the audio rendering engine. What is your planned audio delivery strategy?

GREG LOPICCOLO: We are using a high-performance software synthesizer for two reasons. Reason one is that it basically works on any hardware. We don't have to try to confirm that some specific brand of sound card is present for our game to work. Second, we are trying to do a CD-quality representation of techno or electronic music interactively. Given that, the kinds of timbre manipulations and effects that we need to accomplish this aren't really available using hardware wavetable synthesis right now. The best-sounding solutions that we have available to us now are software synthesizers.

What specific features are you looking for in a software synthesizer?

First, we need very low latency. Our game is a real-time game wherein if you hit a controller button, your brain needs to be convinced that the note sounded when you hit that controller button. So we need a latency of under 20 milliseconds, which is non-trivial to achieve.

Second, we need flexible handling of sample data, like robust high-quality support for stereo samples, looping, and velocity switching. We also need all of the standard professional sampler features like decent, flexible effects manipulation without huge cost. Finally, the most important feature — and the one that has been most difficult to realize — is flexible, high-quality, real-time timbre manipulation capability, like real filters. For techno, it mostly boils down to filters and other modulation sources. With these, you get real-time control over the music that you simply can't reproduce by chopping up samples and blasting them out the DAC. It just doesn't sound the same.

So it is important to have real synthesis for this product, not just PCM sample playback.

Absolutely!

Clearly, there are many methods of synthesis available. What are the most important methods for your application?

For us, particularly because we are directing our early efforts in the direction of techno music, it's pretty much about analog, subtractive synthesis. If we have a good-sounding four-pole resonant filter that we can apply to individual voices, that gets us 80 percent of what we need. We are also working with some physical modeling. For example, the synth that we are using has a good physically modeled guitar-simulator patch. We find it extremely exciting to have real-time feedback and distortion and so forth. The electric guitar is a great sound to be able to have in the game in that it's exciting to play with in real time. But analog synthesis emulation, such as that which can be done with physical modeling is the most important capability for us right now. Emulation is what we are trying to do.

How will the game application interact with the synthesizer to leverage this during game play?

We have real-time, dynamic manipulation of filter settings, like resonance and cutoff, as per standard techno-music parlance. We do a fair amount with effects, especially delay (which is a big deal in techno music), and we use it to great effect. We also employ other modulation effects like low-frequency oscillators for vibrato or filter cutoff. Just having access to those functions as separate real-time modulation sources and destinations gives the music a kind of organic, natural feel that you simply could not get out of a Red Book track or out of chopped up samples.

Which synths have you look at and considered?

We spent some time evaluating the Yamaha XG products, both hardware and software. We've been in touch with Staccato. And then there is Seer Systems' Reality, which is the one that we are currently using for the prototype. It is not really intended as a game engine, but as a low-latency, non-CPU-hogging, high-fidelity, flexible analog synth/sampling package, it's hard to beat.

compatible, for the most part. This allows for the VL-70m to be used as a real-time development platform for the S-YXG playback engine.

Currently, Yamaha focuses exclusively on physically modeling musical instruments. Its VL products use two primary physical models, one of a string and one of a resonating tube. The string model is used for violins, violas, and guitars, while the tube model is used for horns, brass, and woodwinds. Physical modeling, however, is not appropriate for all musical instruments, says D'Amore. For example, a the physical model of a piano would be extremely expensive mathematically because one would not only have to model each individual string, but also the complex mathematical interactions caused by the sound waves emanating from one string acting upon and modifying the sound waves being produced by other strings vibrating in close proximity.

All of the models are developed by Yamaha internally and locked into the product. "You cannot change the core model," notes D'Amore. "The reason we do that is because it is very easy to break a model. If you've done any work with physically modeled products that let you change the core model, you know that 90 percent of what results is unusable for anything. It just won't make any sound. The example I use is that on our develop-

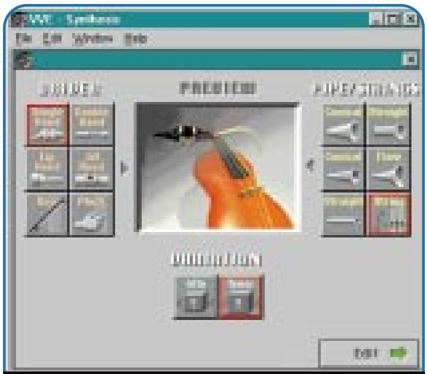


FIGURE 2. *A single-reed driver combined with the string model.*

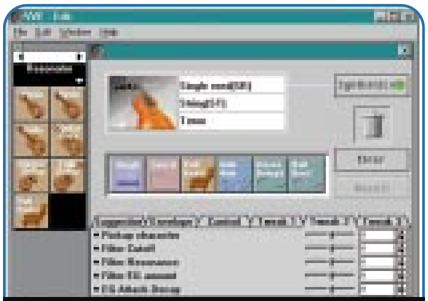


FIGURE 3. *The Resonator Module Palette, the Instrument Editor with six modules loaded and tweakable parameters displayed.*

ment systems, I was able to make an 800-foot-long trombone with a normal mouthpiece. The problem there is that because of the physics involved, there is no way that that it could function. It looked cool, mathematically it was cool, but so what?... Once you get above the core, though, you can take the model and extend it. You can change the envelopes. You can change the materials. You can change how big the mouthpiece is. You can bow the mouthpiece. You can bow the tube. You just can't change the core. We don't allow you to break it that much."

To facilitate this kind of editing, Yamaha provides for free a reasonable, if somewhat obtusely designed, editor for its VL-70 hardware. The editor allows you to match the "Driver" or input components (bow, pluck, lip reed, jet reed, single reed, double reed) with variations of the pipe or string portion (conical pipe, straight pipe, flared pipe, or string) to generate a basic, playable instrument to work with (Figure 2).

Once a basic model is built, you can add a number of other modules to the instrument and tweak their respective parameters. Figure 3 shows the tenor reed string being fed through a single-coil electric guitar pickup, a low-pass filter, a "full pedal" piano resonator, a wah-wah, a stereo delay, and a hall reverb. Also displayed are some of the available tweakable parameters.

It is these exposed parameters that make physical modeling worth implementing for music. If one simply takes a physically modeled instrument and plays it as one would normally play a wavetable sound, the result is remarkably unimpressive. It's only when you begin to feed the instrument large volumes of performance-oriented controller data that the value of the technology becomes apparent. Why? Consider wavetable sounds for a moment. Most wavetable sounds have all of the nuance and gestural components burned into the samples themselves. As a result, simply triggering the sample will produce a reasonable-sounding performance of the instrument's sound. The problem is that it is difficult to manipulate the sample data in real time in appropriate and natural ways. So every time you trigger the sample, you get the same performance, which is extremely unnatural. In contrast, the physical model is more flexible and more demanding. Simply triggering the model without any additional gestural data will produce a very flat performance. Trigger the model and then feed it appropriate gestural data and suddenly the performance comes to life in a much more realistic and responsive fashion than can be achieved with wavetable instruments.

Of course, this kind of power and flexibility comes at a cost. In the physical modeling case, the cost is in CPU cycles. For an average physical model running on a 266MHz Pentium II, developers should expect roughly a six-percent CPU usage. By contrast, the standard software-only wavetable synthesizer that comes with Microsoft's DirectMusic requires approximately 0.17 percent per "dry" voice at 22KHz. Adding reverb will cost an additional 2.8 percent globally. The Beatnik (wavetable) audio engine is about twice as CPU-intensive as the Microsoft synthesizer, and the DLS 1–compatible software synthesizer that ships with RAD Game Tools' Miles Sound System uses approximately one percent of a 266MHz Pentium II per voice on average. (According to John Miles, creator of the Miles Sound System, this figure is averaged to include the possible use of MP3 compression for long DLS voices "and some basic interpolative filtering to clean up resampled instruments.") Still, one percent per voice is still a far cry from what is required to run the physical models on the S-YXG50VL. If you intend to run multiple voices of physical modeling with the S-YXG50 Poly-VL, a 400MHz machine is strongly recommended as a minimum.

D'Amore says that Yamaha's next development step involves trying to get the models to interact properly

FOR FURTHER INFO

Beatnik Audio Engine
http://www.beatnik.com

Harmonix Music Systems' FREQ
http://harmonixmusic.com

Microsoft's DirectMusic
http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/psdk/directx/dmover_6lk3.htm
http://www.microsoft.com/hwdev/audio
http://eu.microsoft.com/directx/overview/dmusic

RAD Game Tools' Miles Sound System
http://www.radgametools.com/mssnew.htm

Seer Systems' Reality
http://www.seersystems.com/products/reality.html

Staccato Systems' SynthCore
http://www.staccatosys.com

Stanford University Center for Computer Research in Music and Acoustics (CCRMA)
http://ccrma-www.stanford.edu

Steinberg's Rebirth
http://www.steinberg.net/products/rebirthpc.html

Yamaha XG
http://www.yamaha.co.uk/xg/index.htm

Yamaha VL Editor
http://www.yamaha.co.uk

with each other. “As true physical modeling implies, you are modeling the physical reality. Physical reality in most circumstances demands physical interaction between models,” he explained. For example, two different models of a trumpet, played together, interact in quite complex ways with one another in the real world. Right now, Yamaha’s synthesizers don’t replicate this effect. As to the suitability of the Interactive Audio Special Interest Group’s (IA-SIG) Interactive 3D Audio Rendering Guidelines, Level 2 (the I3D2 standard is based upon Creative Labs’ EAX technology and adds obstruction and occlusion modeling), D’Amore implied that this would be insufficient to the task. He said that something more like Aureal’s A3D wavetracing technology would be required to achieve the desired effect.

They also have plans for software “S-YXG plug-in” versions of the AN1X (a hardware synthesizer that uses physical modeling to simulate subtractive analog synthesis), a DirectX 7–style six-operator FM synthesizer, and a FS1R-style “singing” synthesizer, which D’Amore says is sadly “Japanese-only for now.”

Recently Yamaha announced that it would be a middleware developer for both Dreamcast and Playstation 2. According to D’Amore, Yamaha is attempting to bring full XG compatibility to both machines, insofar as it is possible, but he gave no firm date for when that goal might be realized.

The Next Big Thing?

Physical modeling is undoubtedly a powerful tool for sound design and composition. The tools to edit and play back (if not create) high-quality models are available today and the advances in sound quality and run-time flexibility are obvious. But the real question is whether the mainstream game development community is ready to make the leap and adopt this new technology. The standard argument against adoption is the amount of CPU power that physical modeling sound requires. In years past, three to six percent of a 266MHz Pentium II per model represented a fairly large performance hit.

The following factors, however, might explain why certain leading-edge developers, such as Electronic Arts and Square, have chosen to take the leap:

- CPU speed is increasing so rapidly today that six percent of a 266MHz Pentium II will likely become meaningless within the next year or two.
- Physical model descriptions (such as Staccato’s DLA files, which are typically 40K or so) are much smaller than a commensurate wavetable sound effect (if creating such effects is even possible).
- Given an appropriate game or music type, it may only require a single, high-impact model to make a significant difference in the overall perceived audio quality of the product.

So, although the technology is still new and developer uptake is just getting underway, the trends seem to be pointing to a bright future for physical modeling and other advanced forms of synthesis. ■

Back

Im leme ti g S bdi isio S face Theo

by Brian Sharp

In my article last month, I explained what subdivision surfaces were and why game developers should be interested in them. I also covered a couple different kinds of subdivision surfaces in their mathematical forms and briefly discussed their benefits and detriments. Most everything was in English, and the rest was expressed using equations. There were no code listings last month, not even a hint of C++, but I promised to discuss an implementation, and so that's the goal of this article. I'll cover a sample implementation of the modified butterfly scheme as discussed in last month's article, complete with a shiny, new demo.

Why the Butterfly?

Last month, I wrote about a number of schemes and those were only the tip of the iceberg, so it's worth spending some time justifying the choice I've made for this implementation. Why use the modified butterfly? To explain my reasoning, it helps to look at more general characteristics of schemes and their advantages and disadvantages. The major differences tend to hinge on whether a scheme is approximating or interpolating.

Approximating schemes have a number of benefits. The surfaces they produce are generally very fair, and they are generally the favored schemes for use in high-end animation. For instance, Pixar uses Catmull-Clark surfaces for their character animation. The downside of approximating schemes are substantial, though. The major one is that because the scheme doesn't interpolate its control net, the shape of the limit surface can be difficult to envision from looking at the control net. The caveat is that as the net becomes denser, the surface will generally be closer to the net. But for games, the net itself won't be tens of thousands of polygons, so the surface can differ substantially from the net.

Interpolating schemes are a different story. They can exhibit problems with fairness, with ripples and undulations over the surface, especially near tight joint areas. Also, they aren't used in high-end rendering quite as much, which can mean that they're the focus of less research. But their major benefit is that the surface is substantially easier to envision by looking at the net. Since the

Brian doesn't need to write a bio for this article, as the Y2K apocalypse will have eradicated all life before the article goes to press. If he survives the fallout, he'll keep right on evangelizing OpenGL and scalable geometry from over at 3dfx Interactive. Wish him a happy new year at bsharp@acm.org.

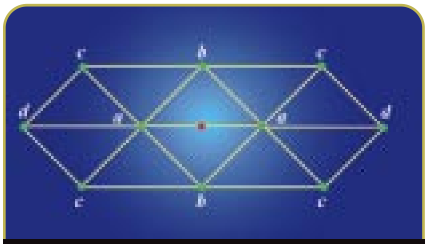


FIGURE 1. *The stencil used for the regular case of the modified butterfly scheme.*

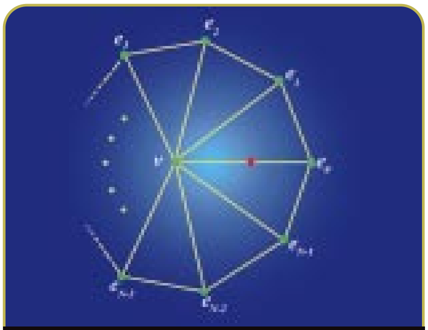


FIGURE 2. *The stencil used for the extraordinary case of the modified butterfly scheme.*

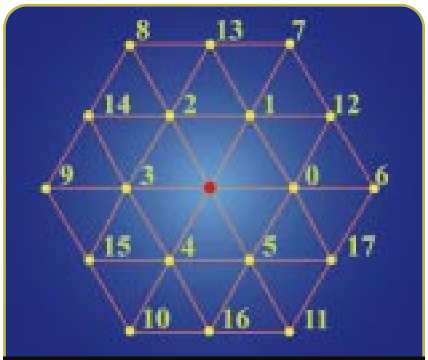


FIGURE 3. *The stencil used for the tangent mask of a regular vertex.*

surface passes through all the net vertices, it won't "pull away" from the net. The fairness issues are the price to pay for this, though. Approximating schemes are fair because the surface isn't constrained to pass through the net vertices, but interpolating schemes sacrifice the fairness for their interpolation.

Nonetheless, I feel that the fairness issues present less of a challenge to intuition than an approximating surface does. For example, in many cases, existing artwork can be used with interpolating schemes with some minor adjustments to smooth out rippling, whereas adapting existing polygonal art to be a control net for an approximating scheme is a much more difficult task.

Among interpolating schemes, the butterfly scheme has a number of things going for it. It's one of the better-researched schemes. It's also computationally fairly inexpensive. Finally, the results of subdivision tend to look good and conform fairly well to what intuition would expect. Therefore, it's my model of choice.

Butterfly in Review

If you haven't already, you probably should read my article from last month's issue for the deeper explanation of the modified butterfly scheme. But in case you haven't, I'll summarize it here. Given a triangular mesh, the control net, we want to subdivide it one step. We first add a vertex along each edge according to specific rules. If the endpoints of the edge are both of valence 6, then we use the stencil in Figure 1, with the weights:

$$a{:}\frac{1}{2},\ b{:}\frac{1}{8}+2w,\ c{:}-\frac{1}{16}-w$$

If one endpoint is of valence 6 and the other is extraordinary (not of valence 6) then we use a special stencil that takes into account just the extraordinary vertex, shown in Figure 2. The weights are computed as follows:

$$\begin{cases} N=3{:}\left(v{:}\dfrac{3}{4},e_0{:}\dfrac{5}{12},e_1{:}-\dfrac{1}{12},e_2{:}-\dfrac{1}{12}\right) \\ N=4{:}\left(v{:}\dfrac{3}{4},e_0{:}\dfrac{3}{8},e_1{:}0,e_2{:}-\dfrac{1}{8},e_3{:}0\right) \\ N\geq 5{:}\left(v{:}\dfrac{3}{4},e_j{:}\left(0.25+\cos\left(\dfrac{2\pi j}{N}\right)+0.5*\cos\left(\dfrac{4\pi j}{N}\right)\right)/N\right) \end{cases}$$

If both endpoints are extraordinary, we average the results of using the above extraordinary stencil on each of them. Again, if this seems a bit too terse, refer to last month's article where I discuss the scheme in substantially more detail.

As far as the butterfly scheme's characteristics, it's interpolating because points in a control net also lie on the limit surface — the subdivision process doesn't move existing vertices. It's also triangular as it operates on triangular control nets. It's stationary as it uses the same set of rules every time it subdivides the net, and uniform because every section of the net is subdivided with the same set of rules.

One aspect of the scheme that I mentioned last month but didn't define was the tangent mask of the butterfly scheme. This is the mask used to compute the tangent vectors explicitly at a vertex, which we use to find the vertex normals. The mask is large and therefore may look intimidating, but it's just a bunch of numbers, and a few multiplications and additions later, we've got the answer.

For regular vertices, the process involves the 1- and 2-neighborhood of the vertex (so it uses vertices that are one and two steps away.) Between both neighborhoods, there are 18 vertices, and so the scalars, corresponding to the indexing shown in Figure 3, are:

$$l_0=\left\{16,-8,-8,16,-8,-8,-\frac{8\sqrt{3}}{3},\frac{4\sqrt{3}}{3},\frac{4\sqrt{3}}{3},-\frac{8\sqrt{3}}{3},\frac{4\sqrt{3}}{3},\frac{4\sqrt{3}}{3},1,-\frac{1}{2},-\frac{1}{2},1,-\frac{1}{2},-\frac{1}{2}\right\}$$

$$l_1=\left\{0,8,-8,0,8,-8,0,-\frac{4\sqrt{3}}{3},\frac{4\sqrt{3}}{3},0,-\frac{4\sqrt{3}}{3},\frac{4\sqrt{3}}{3},0,\frac{1}{2},-\frac{1}{2},0,\frac{1}{2},-\frac{1}{2}\right\}$$

Multiplying the vertices by l_0 and l_1 gives us two different tangent vectors, the normalized cross product of which is our normal. For extraordinary vertices the normal is actually easier to find, as it depends only on the 1-neighborhood of the vertex. The two tangent vectors in this case can be found as:

$$t_0=\sum_{i=0}^{N-1}e_i\cos\frac{2i\pi}{N},\quad t_1=\sum_{i=0}^{N-1}e_i\sin\frac{2i\pi}{N}$$

Here, t_0 and t_1 are the tangents, N is the vertex valence, and e_i is the ith neighbor point of the vertex in question, where e_0 can be any of the points (it doesn't matter where you start) and the points wind counterclockwise. Crossing the two resulting vectors and normalizing the result produces the vertex normal.

Implementation: The Big Idea

The idea behind our implementation is, at a high level, very straightforward. Given one control net, we want some piece of functionality that can take that net and output a more complex net, a net that has been advanced by a single subdivision step.

That sounds easy enough, right? Unfortunately, that description doesn't translate very directly to C++ code. So we need to define some of our terms and be more specific. First of all, what's a control net? We know what it is conceptually, but what kind of data structure is it and how is it manipulated? After that, of course, we need to define that "black box" bit of functionality that subdivides the net, and quantify how it works.

To establish our control net data structure, we start with nothing and build our way up as needed. So, the first thing we need is the base representation that will eventually pass into OpenGL. That's just a few arrays. We need an array for our vertices, our texture coordinates, and our colors. Furthermore, we'll need an array of indices into those arrays to define our faces; every three indices defines a triangle.

If we can do our tessellating with no more than that, then that's great. But chances are we're going to need to keep around more information than just that. The important thing is that whatever information is added to the data structure needs to be renewable. That is, since the process is iterative, the information we have in the simpler net coming in must also exist in the more complex net coming out, so that we can feed the complex net back in to produce an even more complex net.

It's worth asking why we'd need more information than just the vertices and faces. After all, if we need to determine whether one vertex is connected to another by an edge, we can determine that by looking through the faces. Or if we need to find all the edges, we could just do that by running through the face list, too. The problem here is in the running time of the lookups. When we're subdividing an edge, we need to find out a lot of information about nearby vertices and faces, and we'd like it to be as fast as possible. Regardless of the processor speed, looking through all the faces to find a vertex's neighbors will be slower than if we have that information available explicitly. This is because looking through the list of faces takes O(*F*) time, where *F* is the number of faces. On the other hand, if we have the information stored explicitly, it only takes O(1) time — constant time. That means that as we add more faces to the model, the former solution takes longer, whereas the latter remains the same speed.

We don't have the information we need to decide what else to add to the control net data structure, so we'll work on the procedure for subdividing a net and add data to the control net as necessary.

The Subdivision Step

Our task, then, is this: given a net, we need to subdivide it into a more complex net. Working from the modified butterfly rules, this is fairly straightforward. We need to add a vertex along each edge of the net. Then we need to split each face into four faces using the new vertices.

The first step, adding new vertices along each edge, tells us quite a bit about some more information we'll need in the control net data structure. There's no fast and simple way to find all the edges unless we store them explicitly. An edge needs to be able to tell us about its end points since we need to use those in the butterfly stencil for computing the new vertex. Furthermore, the stencil extends to the end points' neighbors, so the end point vertices need to know about the edges they're connected to.

The second step, breaking existing faces into new faces, requires that the faces know about their vertices, which they already do. The faces also need to know about their edges. While they could find this by asking their vertices for all their edges and fishing through them, that requires a fair amount more work for every lookup, and so we'll explicitly store with each face the information about its edges, too.

That increases the load a fair amount. Our data structure now has arrays of vertices, edges, and faces. Vertices know about their edges, edges know about their vertices, and faces know about their vertices and edges.

Graphs and Subdivision

It's worth noting that the data structure we're working with is nothing new and unusual. It's a specific example of a general data structure known simply as a graph. A graph is anything composed of vertices connected by edges. For instance, a linked list and a binary tree are both special kinds of graphs.

What makes our problem a little tougher than, say, writing a singly-linked list class is that the graph of vertices in a model is considerably more complex than the graph of nodes in a linked list. First, the nodes in a linked list have a single edge coming out of them (pointing to the next node) and one coming in (from the previous node.) Our graph has six edges coming into each regular vertex and potentially many more than that for extraordinary vertices.

Furthermore, in the case of a singly-linked list or a binary tree, the edges have direction. That is, you don't generally walk backward through the list or up the tree. Furthermore, these structures are acyclic — there are no "loops" in them — so from a given vertex, there's no path that leads back to the same vertex. In our case, the edges are undirected. You need to be able to traverse every edge in both directions.

Discussing graphs in this context is really just "interesting facts" rather than being a crucial contribution to our implementation, but it confirms what we already know: our data structure is complicated. The one saving grace is that our algorithm is based on locality, so we don't need to worry about traversing huge distances across the graph to find information we need to subdivide. This is one benefit of using a scheme with minimal support. A scheme with much broader support would be computationally much harder to evaluate, and hence be much slower and far more difficult to implement.

It also confirms the direction we're taking to implement the data structure — it's based wholly on locality so that the time it takes to find one vertex given another is proportional to the number of edges between them. There are other ways of representing graphs for the myriad applications that have different requirements. Cormen and his co-authors (see For Further Info box, p. 45) provide an excellent introduction to graph theory.

Control Net Details

So we know the data we need in our control net data structure and we know the steps the tessellation needs to execute. We're ready to dig into the lower-level implementation details. First, we'll go back to the information in the control net structure and look at how it should be laid out.

Listing 1 shows the layout of the data. There tend to be two schools of thought on data layout. One method is dubbed the "structure of arrays" (SOA) and the other is the "array of structures" (AOS). The idea is that the SOA method stores multiple parallel arrays whereas the AOS method stores all the data interleaved in the same array. I've personally never run into a situation where the two approaches differed greatly in speed, and so when I lay out data I generally try to blend the two approaches for clarity's sake. That's why some of the data in the listing is shown as separate arrays of base types and some are stored as arrays of small objects.

LISTING 1. *The data used to represent the control net.*

```
class ButterflySurface
{
public:
  ...

protected:
  ...

  // Information about the vertices
  int numVerts;
  int vertCapacity;
  float* verts;
  float* vertNorms;
  VertexEdges* vertEdges;
  float* texCoords;
  float* colors;

  // Information about the faces;
  // all faces are triangles.
  int numFaces;
  int faceCapacity;
  int* faces;
  int* faceEdges;

  // Connectivity information,
  // needed for tessellating.
  int numEdges;
  int edgeCapacity;
  ButterflyEdge* edges;
};

// Classes used in control net storage.
class VertexEdges
{
public:
  VertexEdges();
  VertexEdges(const VertexEdges& source);
  VertexEdges& operator=(const VertexEdges& source);
  int numEdges;
  int edges[MAX_VERTEX_VALENCE];
};

class ButterflyEdge
{
public:
  bool operator==(const ButterflyEdge& cmp) const;
  bool operator<(const ButterflyEdge& cmp) const;
  int v[2];
};
```

The vertices are stored in OpenGL-friendly arrays. While OpenGL allows for interleaved arrays, many applications tend to store their data in parallel arrays, and that's why I choose to do so as well. The vertices, texture coordinates, normals, and colors each have their own arrays. These arrays are dynamically grown; when I need to add another vertex and there isn't sufficient room, I allocate new arrays that are twice the size of the current ones and move the data into the new arrays. This strategy amortizes the cost of memory allocation and is one I use for most of my memory management.

Each vertex also has a VertexEdges associated with it. VertexEdges keeps track of the edges that the vertex is a part of. Following the theme of making lookups as fast as possible, the edges are stored sorted by winding order, so each successive edge in the array is the next edge in counterclockwise winding order from the previous edge.

The edges themselves prefer the AOS format. Each edge is stored as nothing more than two indices into the vertex arrays. Adding another nitpicking detail, I sort the indices by value. It comes in handy as there are many cases where I can skip a conditional by knowing that they're in sorted order.

The faces are stored simply as an array of indices into the vertex arrays, where every three indices defines a triangle. Since the control net is totally triangular, I don't need any complicated support for variable-sized faces.

That's it for the storage of the control net. Now we need to understand the details of the tessellation process.

Subdivision Step Details

As mentioned earlier, the subdivision step consists of subdividing edges and then building new faces from them. The top-level function that does this is shown in Listing 2. For the edge subdividing, I iterate over the edges. At each edge, I check the valences of the end point vertices to determine which subdivision rules to use. Upon deciding that, I apply the rules and produce the new vertex. It's then added to the end of the vertices array.

Furthermore, the edge is split into two edges. One of them uses the slot of the old edge, and one of them is added to the back of the edge array. For use in building the faces, I keep two lookup tables. One maps from the old edge index to the index of the new vertex I just created. The other maps from the old edge index to the index of the new edge that I just added.

Building the faces is somewhat more involved, as it requires a fair amount of bookkeeping when creating the four new faces to be sure that they're all wound correctly and have their correct edges. For each face, I have the corner vertices and the edges. From the two lookup tables I created while subdividing edges, I also know the new vertices and new edges.

I shuffle all that data around to get it in a known order so that I can then build faces out of it. I also end up adding three more edges connecting the new vertices inside the triangle. Those new edges need to be added to the new vertices' edge lists, and they need to be added in the correct winding order. This isn't much code, but it's tricky and bug-prone.

Using this function, I can iterate over that as many times as I like. Each iteration increases the polygon count by a factor of four. When I decide to stop, only then do I need to worry about calculating vertex normals. Iterating over the vertices with the modified butterfly tangent mask finds those handily.

LISTING 2. *The top-level function used to tessellate a control net.*

```
// This tessellates the surface.
void ButterflySurface::tessellate()
{
  // Loop controls.
  int x;

  for (int level=0; level<maxRecursion;
                                level++)
  {
    // This is how we later find the new
    // vertices created along edges.
    int* edgeVertMap = new int[numEdges];
    for (x=0; x<numEdges; x++)
    {
      edgeVertMap[x] = -1;
    }

    // This is how we find the new other
    // half-edge made when the edge is
    // split.
    int* edgeEdgeMap = new int[numEdges];
    for (x=0; x<numEdges; x++)
    {
      edgeEdgeMap[x] = -1;
    }

    tessellateEdges(edgeVertMap,
              edgeEdgeMap);
    buildNewFaces(edgeVertMap,
              edgeEdgeMap);

    delete[] edgeVertMap;
    delete[] edgeEdgeMap;
  }

  // Only at the end here do we generate
  // our normals.
  generateVertexNormals();
}
```

Colors and Texture Coordinates

The previously described procedure finds the vertices and normals, but not the colors or texture coordinates. These deserve their own discussion. Colors are nice because they can be interpolated using the same scheme as the vertices. If the butterfly scheme produces smooth surfaces in XYZ space, it will also produce smoothness in RGBA space. It's certainly possible to linearly interpolate the colors. That will result in colors that don't change abruptly, but whose first derivative changes abruptly, resulting in odd bands of color across the model, similar to Gouraud interpolation artifacts.

Texture coordinates are a somewhat more difficult problem. Current consumer hardware interpolates color and texture over polygons in a linear fashion. For colors, this isn't what we generally want: Gouraud interpolation of color exhibits significant artifacts. But for texturing, it is what we want. The texture coordinates should be linearly interpolated, stretching the texture uniformly across a face.

Therefore, when I interpolate texture coordinates during subdivision, I just linearly interpolate them. Furthermore, higher-order interpolation doesn't necessarily make sense at all, as different faces of the control net might have totally different sections of the texture, or even have totally different textures mapped onto them. While the data structure doesn't currently support this (vertices would need to be capable of having multiple sets of texture coordinates), it could certainly be desirable. In this case, neighboring vertices' texture coordinates are in totally different spaces, so interpolating between them doesn't make sense.

So, I'll stay with linear interpolation for texture coordinates. In terms of elegance, this method is a little disappointing. If we interpolated everything using the modified butterfly scheme, we could treat vertices not as separate vertex, color, and texture-coordinate data, but as one nine-dimensional vector, (x,y,z,r,g,b,a,u,v), and just perform all the interpolation at once. Alas, in this case, elegance needs to take a back seat to pragmatism.

Now we know how to start with a control net and step forward, producing increasingly detailed control nets, all the while keeping our data structures intact and keeping our vertices, colors, and texture coordinates intact, and generating normals for the finished model. What else is there left to cover?

Animation

While it's beyond the scope of this article to describe how you might implement a full animation system that uses subdivision surfaces, it's worth describing how subdivision surfaces and animation can coexist. If your game is one that stores the animated model as a series of full models, clearly you don't even have to think about it — subdividing those individual meshes will just work.

Skeletal animation is a somewhat more interesting problem. One of the nice things about subdivision surfaces is that a skeletal animation system should be able to transform the control net before subdivision, saving you the cost of multiple-matrix skinning on the high-polygon final model. This does have some downsides, though. Depending on the model and a host of other factors, the skeletal animation might cause the model to flex in strange ways or to exhibit increased rippling or unfairness.

The other downside is that it doesn't allow your application to take advantage of forthcoming hardware that supports skinning on the card. Depending on the speed of that skinning, though, and on how many times you subdivide the model, the savings of you doing a reduced number of transforms may or may not be worth the loss of offloading.

Adaptivity

Since this is a scalable geometry solution, it's worth asking if we can adaptively subdivide based on curvature or distance to the camera. In my previous articles on tessellating

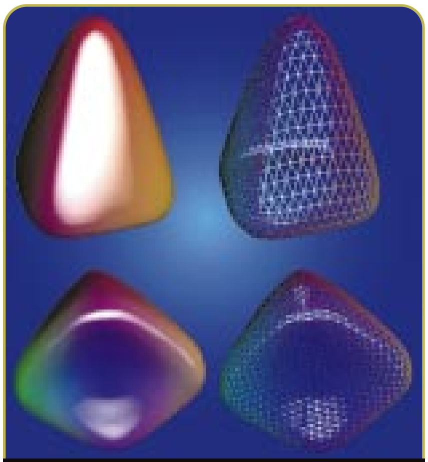


FIGURE 4. *Subdivision steps of a colored shape in the demo.*

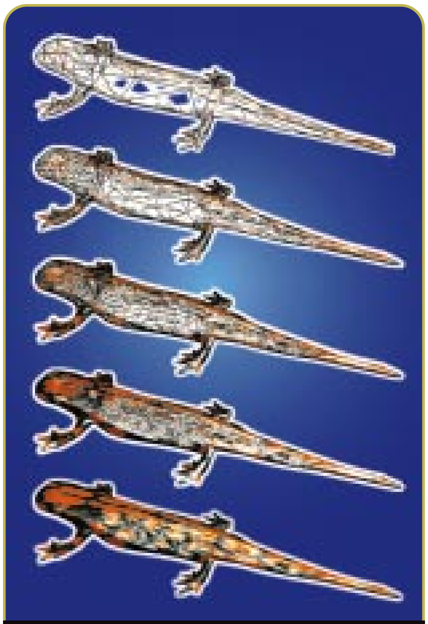


FIGURE 5. *Subdivision steps of a salamander model in the demo.*

Bézier patches ("Implementing Curved Surface Geometry," June 1999, and "Optimizing Curved Surface Geometry," July 1999) such adaptivity was a major focus.

The problem with adaptive solutions for subdivision surfaces is that, unlike patches, subdivision surfaces don't easily expose a closed-form parameterization. The only easy way to tessellate them is through recursion. So we rely on the fact that as we recurse, we're converging on a limit surface. And no matter how we tessellate, we should be converging on the same limit surface.

If we tessellate adaptively, we've changed the control net. Some of the net might be at a higher level of tessellation than the rest. And so we've broken the rules, and our net is no longer converging on the same surface. This is a worst case scenario for scalable geometry — it produces a "popping" that you simply can't avoid, since the underlying surface is now fundamentally different.

Furthermore, although this could probably be dealt with somehow, would it be worth it? Consider that a game probably won't be subdividing the control net more than four times. If your original net is, say, 1,000 polygons, four subdivision steps bring it to 256,000 polygons. The span of low-end to high-end machines isn't yet quite that large. So the end result of an elaborate adaptivity scheme would just be a model that was subdivided three times in some areas, maybe four in others: a whole lot of work for negligible benefits.

If you're using subdivision schemes for characters, then unless your characters are gigantic, adaptivity based on distance from the camera won't be worth much, either. Plus, characters tend to be fairly uniformly curved; most of them don't have large flat sections and jagged spikes in other areas. Therefore, in the end, you might be able to squeeze some benefits out of an adaptivity scheme, but the amount of work necessary to do so is fairly daunting. It's probably sufficient to pick a subdivision level based on distance to the camera and field-of-view angle and tessellate to it.

The Demo and Further Work

As promised, this month is accompanied by a demo built off the sample implementation provided above. A few screenshots are shown in Figures 4 and 5. The demo is available at my web site (see Further Info. box) and comes with source code and a couple of sample models.

I'll freely admit that the demo is not at the point where you could drop it straight into your game and witness a stunning transformation (unless shiny salamanders are exactly what your game needs). There's a good deal more to be done with the demo. For starters, it's worth asking what to do when even the base control net is too dense. If a character is far away from the camera, maybe you'd only like to draw a 200-polygon version? In that case, integrating a separate mesh-reduction algorithm that you apply to the simplest net when needed could solve the problem nicely.

Another issue that the demo doesn't address is the question of caching. I currently regenerate the subdivision from the base net every frame. Is it worth caching subdivisions? On one hand, it could make things faster, but if the models being subdivided are characters, then the animation probably makes caching less useful, since the model you created in one frame isn't in the right position by the next frame.

Whether or not the modified butterfly scheme is the right one for you, this demo should provide a decent starting point for experimentation. Hopefully, between these two articles, I've given a solid overview of subdivision surfaces, and maybe even gotten somebody interested in using them in a game or two. Questions and comments are heartily encouraged, and in the meantime, I hope to find myself amazed by the next generation of fully scalable, beautiful games. ■

FOR FURTHER INFO

The demo and other resources are available at my web site:
http://www.cs.dartmouth.edu/~bsharp/gdmag

Additional Resources

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

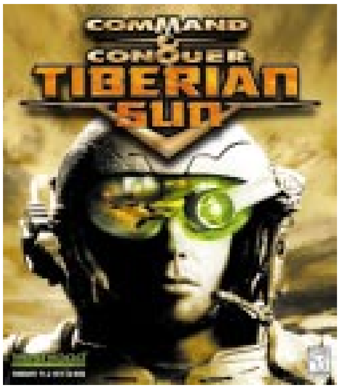
Zorin, D. "Stationary Subdivision and Multiresolution Surface Representations." Ph.D. diss., California Institute of Technology, 1997. (Available at ftp://ftp.cs.caltech.edu/tr/cs-tr-97-32.ps.Z)

Zorin, D., P. Schröder, and W. Sweldens. "Interpolating Subdivision for Meshes with Arbitrary Topology." *Siggraph '96*. pp. 189–192. (Available from ACM Digital Library.)

West ood St dios TIBERIAN SUN

by Rade Stojsavljevic

Ever since the release of Westwood's Dune 2 in 1992, real-time strategy (RTS) games have become the hottest-selling computer games around. Countless RTS games were released soon afterward including Command & Conquer (C&C), Red Alert, Warcraft II, Age of Empires, and Total Annihilation. These games have propelled the genre to new heights and have drawn an increasing number of fans.

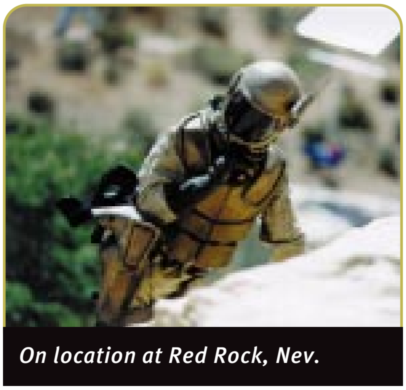


After the success of C&C and Red Alert, the team at Westwood Studios started work on Tiberian Sun, the sequel to C&C. To build the game, we assembled a team that consisted of veterans from C&C and Red Alert along with a

Rade Stojsavljevic was the producer for Tiberian Sun and is currently working on an expansion pack called Firestorm. Before coming to Westwood, he worked on military simulations and adventure games at various small development houses. When he's not out getting doughnuts to bribe the team with, you can usually reach him at rade@westwood.com.

Umagon prepares for a take.

Chandra, McNeil, and Brink pose on the Kodiak Bridge.



On location at Red Rock, Nev.

couple of new faces, including me. We started with the goal of taking what made C&C fun and expanding it even further.

To begin the development process we reviewed what makes a great RTS game and came up with one answer: tactics. Westwood doesn't build games based on a specific technology and we never sell technology over the game play. We have a firm belief that a great strategy game must have interesting, fun, and new tactics that afford players a multitude of unique ways to play a game.

We wanted TIBERIAN SUN to appeal to a broad audience, yet also appeal to core game players and fans of the series. Towards this goal, we continued to apply a "wide and deep" approach to designing the tactics we created. Wide and deep essentially means a nice assortment of diverse yet readily apparent tactics that, under the surface, contain an even greater number of tactics. With this approach, you can provide first-time players with a number of different things to do while letting more experienced players discover new and advanced tactics on their own. These design goals made working on the game more challenging — as if being the biggest project in Westwood Studios' history wasn't enough.

What Went Right

1. MAINTAINED C&C STYLE OF GAME PLAY. One of the most difficult tasks we had to overcome during the development of TIBERIAN SUN was to maintain the feel of the original. When making a sequel, the question that always has to be answered first is, How far do you stray from the original game to make it compelling, yet still familiar? The intent with TIBERIAN SUN was to maintain, as much as possible, the feeling of the original while providing new and interesting tactics for players to master. To aid in this goal, when adding a new feature we asked the questions, "Is this consistent with COMMAND & CONQUER?" and "How can we make it easier and even more exciting?"

In this area, it really helped to have a development team that worked on the previous games. They were able to draw from previous experiences to create a consistency in the game dynamics. This gave the team a great deal of independence since everybody already had a good idea of how the game was supposed to look, play, and feel.

The main areas we focused on in order to be consistent with previous games were the user interface and unit behavior. We knew we had to keep the sidebar metaphor for unit construction but we wanted to update it to accommodate new features, such as unit queuing, waypoints, and power/energy control. For unit behavior there was a set of rules that we had to conform to, specifically how a unit deals with player commands so that its internal logic never overrides a player's orders. One of the times we tried to change the rules was when harvester threat-avoidance logic was introduced. I remember hearing lead designer Adam Isgreen screaming at his computer when his harvesters refused to obey his orders to retreat. We decided to scrap that idea shortly afterward.

It was important for the overall visual presentation of the game to bear a resemblance to its predecessors in order to maintain a consistent artistic style. We decided to alter the perspective slightly, rotating the camera to create a three-fourth isometric perspective that afforded a better sense of depth and realism in a 3D perspective. It was at this point that we decided not to use a polygonal engine since it wouldn't be possible for us to keep the system requirements low enough to achieve the mass-market appeal that we wanted. Also, at the time we planned to release TIBERIAN SUN, 3D accelerator cards and systems weren't fast enough for us to maintain the visual detail we wanted for the hundreds of units and structures on-screen at once.

2. WORKING ON A SEQUEL TO A SUCCESSFUL FRANCHISE. Being the fourth RTS game Westwood has done, there were a lot of lessons learned that the team was able to carry forward into TIBERIAN SUN. First, we had an established and

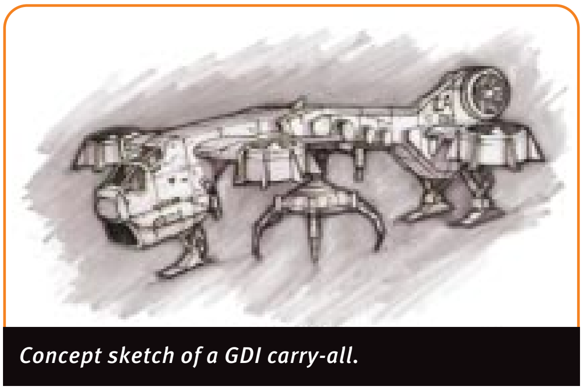
TIBERIAN SUN

Westwood Studios
Las Vegas, Nev.
(702) 228-4040
http://www.westwood.com

Release date: September 1999
Intended platform: Windows 95/98/NT 4.0
Project length: 36 months
Team size: 25 full-time, 15 part-time developers
Critical development hardware: Pentium Pro and Pentium II machines, 200 to 450MHz dual-processor with 128 to 256MB RAM, Creative Labs sound cards, Windows 95/98/NT, SGI O2 workstations, BlueICE accelerators
Critical development software: Microsoft Visual C++, Lightwave, 3D Studio Max, Discreet Flint, Adobe Photoshop, Adobe After Effects, Adobe Illustrator, Avid Media Composer, Filemaker Pro, Deluxe Paint

streamlined user interface. This user interface has been a cornerstone of Westwood RTS games since Dune 2 and we've been gradually improving it ever since. Anyone who has ever played a Westwood RTS is immediately familiar with the controls and can jump right into the action. Additionally, the interface is simple and intuitive enough to let new users become comfortable with it in a short time.

An Orca carry-all transports a hover MLRS.



Concept sketch of a GDI carry-all.

Another nice benefit of making a sequel is that we had a set of basic features we knew worked based on previous games. These provided a solid foundation that could be expanded upon and modified as needed. We started with features from the previous games that we knew we wanted and updated them to fit a world that was 30 years in the future. Tanks evolved into two-legged mechanized walkers, soldiers could now use drop pods launched from space, and cloaking technology advanced to yield a stealth generator that hid many units and buildings at once.

When it came time to create the story, we already had the basic framework in place. There was a very rich and fascinating world to draw upon when creating new characters for this story. The one difficulty encountered was making sure the story could stand up on its own and be accessible to new players without subjecting players familiar with previous games to mind-numbing exposition. To solve this problem, we set the story 30 years after the end of the original, which provided an opportunity to create an outstanding introduction that showed players what had been going on in the world.

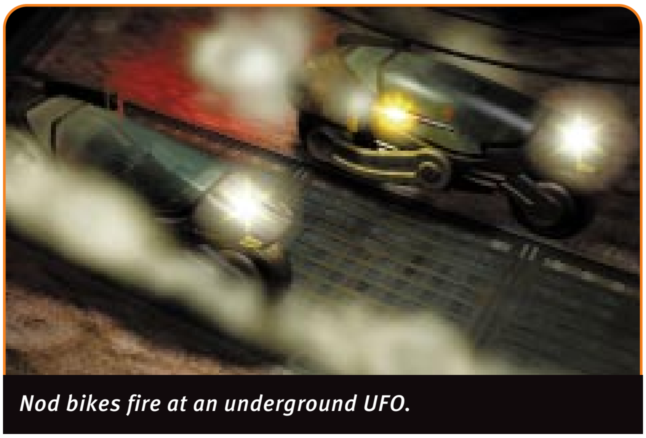
3. Team Experience and Cohesion. The Tiberian Sun development team is one of the most experienced and professional teams I've ever had the privilege of working with. For many of the team members, this was the fourth RTS game they had done (the previous being Dune 2, C&C, and Red Alert). This level of experience was key in allowing the team to conquer all the obstacles thrown in their path. Even though I had worked on half a dozen titles before I started on Tiberian Sun, at first it was a little unnerving for me to be working with a team of this caliber.

Several members of the programming team had worked together on previous Westwood RTS products and were accustomed to each other's coding styles. New programmers were quickly assimilated into the team and were able to adapt well. The coding rules and Westwood libraries allowed the programmers to familiarize themselves with each other's work with minimal difficulty.

The designers had worked on previous RTS games and were very familiar with the universe before we started the project. This saved several months since no one had to familiarize themselves with anything except the design for Tiberian Sun. The tools used were derivatives of the C&C and Red Alert editors, which also minimized the ramp-up time required before they could produce missions. The designers worked well together and were friends; something that helped a lot when there were differences of opinion. It proved to be very beneficial to know that you could argue your point and not have to worry that the person you were arguing with would hold a grudge afterward.

Without the technical knowledge and creativity of the artists on the project, we would have suffered a great deal of pain when integrating artwork. Like most projects, Tiberian Sun had a specific set of technical criteria that had to be satisfied when creating art for the game engine. On this front, we reaped the rewards of having artists who had done it all before. They had worked with our programming team and knew the tools well enough that they were able to head off potential problems before they could get out of control. The cinematic artists had much of the same experience; they didn't have as many technical restrictions as the in-game artists, which allowed them to be able to express unbridled creativity. The cinematic artists didn't have to deal with frame limitations or palettes. Also, compared to previous games, the movie player in Tiberian Sun allowed for full-resolution movies (as opposed to previous games where every other line was cut out) using 24-bit color depth and a 15FPS frame rate. I still remember the first time we saw the movie in which the Mammoth Mk. II laid waste to an entire Nod base by itself; it left everyone in the room speechless.

The final piece was the management team. Under executive producer Brett Sperry's strong leadership, we established systems to deal with routine tasks, facilitated communication between the teams, and were able to avoid a lot of problems early on. Brett



Nod bikes fire at an underground UFO.

Nod bikes flee from the ensuing explosion.

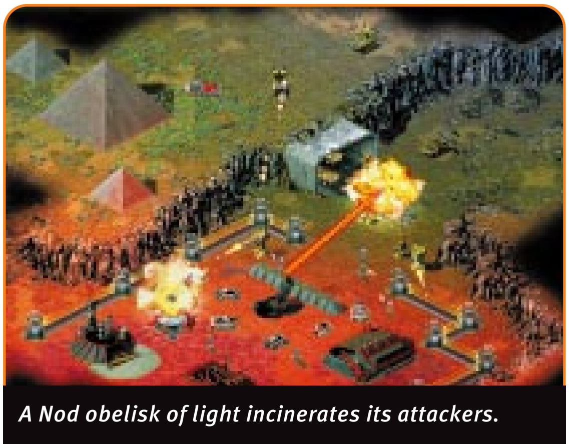
has always been very protective of the C&C franchise and with TIBERIAN SUN, his clear and consistent vision of where the game should be was absolutely critical to the project.

4. BALANCING PROCESS. Balance is one of the things that can make or break a RTS game. It's one of the hardest things to do on the design side of the product since you're essentially trying to optimize an equation with dozens of independent variables. If you get it wrong, you'll have a boring game and a horde of disgruntled fans cursing your name forever. When the issue of balancing comes up, you'll often hear about the "rock-paper-scissors" idea, but I like to think of it more in terms of a chess game. You've got a lot of different pieces, each with a unique function and set of strategies that takes a long time to master.

Having made several RTS games before, the team knew how to balance a game. We started with two approaches: one scientific and one artistic.

Using the scientific approach, we started with the relatively simple idea that in a steady state units with an equivalent cost should do equivalent damage to one another. The basic idea is that if I have $1,000 worth of units and you have $1,000 worth of units and they fight, the fight better be really close. From here, we kept adding variables until we had a relatively playable game.

The next step was a lot more artistic and was where experience really paid off, keeping the team from long periods of fumbling around blindly. We played countless games with each of us championing one side vs. the other, carefully noting how effective units and tactics felt against one another. We would get together after each game to compare notes, argue our points, get into fights, and then make one change at a time to the game and try it again until we were all satisfied with the results. The whole process took about three months for TIBERIAN SUN, compared to six months for C&C and four months for RED ALERT. Even after the countless games we played against one another, we still got into shouting matches during close multiplayer games. When this happens, you know you've got a winner on your hands.



A Nod obelisk of light incinerates its attackers.

GDI forces destroy a vital Nod caravan.

5. MISSION DESIGN. Mission design is one of the most important elements of RTS games. Based on experience with previous games, Westwood has established a series of processes that are used whenever a mission is created. We've designed these processes to foster creativity, maximize efficiency, and promote communication between the design, programming, art, and management groups. This process has been refined on every project and we've taken it to the next level with the upcoming FIRESTORM add-on.

The process begins with a mission design proposal submitted to the lead designer and producer. The proposal is a two- to three-page document that contains summary information about the mission such as name, side, difficulty, map size, mission type, and so on. The mission briefing is included along with a description of what the briefing movie should be and all of the critical information that must be revealed to the player. Mission objectives are listed as they would appear in the game, along with specific information on how to achieve the objectives. Win and lose conditions are created, as well as descriptions of the victory and defeat movies that play at the end of a mission. The last things included are all of the new voice and text messages used in the mission.

GDI Titans lay waste to the Nod base.

Once this proposal has been approved, the map for the mission is sketched out on paper. We've found that this process can save a great deal of time since it eliminates distractions and allows the designers to get an overall view of the map quickly. When the designers finish sketching their mission, they proceed to the editor and begin to create the basic battlefield. Terrain is laid down first, followed by buildings, roads, trees, and pavement.

The final step to complete a mission is to take a map and add scripting, which takes approximately two-thirds of the time to create a mission. One of the great things about TIBERIAN SUN is that the editor is tied directly into the game, which allowed designers to switch rapidly between the editor and the game. This feature also proved to be a liability, however, because if a bug appeared that prevented the game from running, we couldn't run the editor, either.

TIBERIAN SUN features a good blend of production (such as building bases) and non-production missions that keep the pace of the game interesting and challenging. We tried not to do the same mission twice and added variety by combining mission types into non-production/production missions that switch from one to the other when players reach specific objectives. Branching missions were added to give players the option of completing sub-missions before they tackled the main objective. By playing sub-missions first, the player makes the final objective easier and it gave the designers added granularity when creating the difficulty levels for the game.

What Went Wrong

1. UNREALISTIC EXPECTATIONS. The degree of hype and expectations that TIBERIAN SUN had to fulfill was staggering. We had a team of experienced developers who wanted to beat their own expectations while simultaneously building a game that would be everything the fans of the series expected and more. This was not a realistic goal since it's just not possible to make something that will meet everyone's expectations.

Concept sketch of a GDI Titan.

One of the things that we did not do was explore all of the new features to their logical conclusions. This would have allowed us to do a lot more with a smaller feature set and provide an even better game. A perfect example of a feature that was begging to be used more is the dynamic-battlefield concept. The basic idea behind the dynamic-battlefield concept is that players' actions alter the battlefield. For example, a player could set fire to trees to burn a path into an enemy's base. We wound up cutting this particular feature because it caused path-finding problems. Also, battles with heavy weapons would cause cratering of terrain which hindered unit movement.

We could have used it to create more new strategies for players, and since it was one of the more expensive features in the game, we could have squeezed a lot more use out of it.

Trying to fill the shoes left behind by RED ALERT proved to be daunting. If you had asked a dozen people what they expected out of TIBERIAN SUN before it was released, you would have heard a dozen different answers. We devoted a lot of effort to add as many features into the game as possible instead of just trying to make the best game we could. Getting into a feature war is one of the worst things that can occur during development because it siphons effort away from adding the "fun" to the game.

2. FEATURE CREEP. TIBERIAN SUN started strong and we developed a robust and large feature set we intended to fulfill. The project started smoothly, but as we progressed, the temptation to add new features not included in the design document grew. These features arose out of shortfalls in the original design, omissions from the original design, and input from fans.

Everybody stresses the importance of working off of a design document and not deviating from it. Unfortunately, this just isn't realistic since every product evolves during the course of development and sometimes the original design proves to be lacking. A team has to be able to incorporate new ideas during development if the final project is to be better. However, the flip side of this idea is that the team must be able to cut features diplomatically when it is in the best interest of the project.

TIBERIAN SUN's development had many challenging moments when features had to be cut for one reason or another. A perfect example of this was the ability to order a limited number of units through a drop-ship loading screen before a mission. This sounded like a great idea on paper and we had already coded it and incorporated it into the game. It wasn't until we actually played with it that we realized it just didn't fit and had to be removed.

Looking back at the project, I think we could have been more aggressive in cutting or changing certain features to make sure their returns were really worth the development investment. I'm a firm believer in the idea that less is more and that fewer but more fully developed features are the way to go. If a feature isn't amazing, you should cut it or make damn sure it becomes amazing before you ship the product.

3. POST-PRODUCTION COMPLICATIONS, COMPOSITING WOES. TIBERIAN SUN features the most complex and highest-quality cinematic sequences Westwood has ever done. These movies help drive the story elements forward. However, these movies came at a very high price.

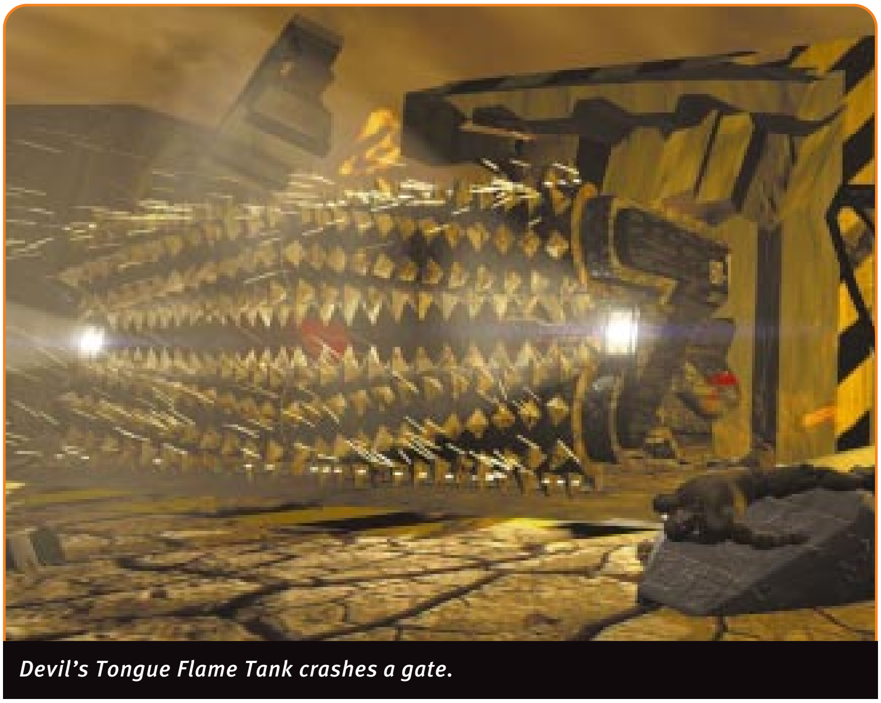
Westwood has a soundstage with a bluescreen and in-house post-production capability that allowed us to handle the entire production ourselves. We've done several different projects with video, including RED ALERT, DUNE 2000, and RED ALERT RETALIATION for the Playstation. Based on these past experiences, it was decided that we would push the limits of what we could do in TIBERIAN SUN.

We started by fully storyboarding every scene in the script. From the storyboards, we built concept sketches of the major sets to be constructed (practical as well as computer-generated) and proceeded to build the sets. Before the shoot there was a three-month lead time for our team of six 3D artists to build the sets. We wanted to have the sets 100 percent complete so we would have camera and lighting information to match up with the live actors.

For various reasons, the pre-production for TIBERIAN SUN was much shorter than it should have been. If you've ever worked in film or television production, you've probably heard the phrase "we'll fix it in post." Believe me when I say there's a reason why this lit-

A disc thrower waits for reinforcements.

A hover MLRS fires a volley.



Devil's Tongue Flame Tank crashes a gate.

Drop-pod infantry surveys the battlefield.

tle phrase can spook even the most veteran members of any production crew. Anything you have to fix after the fact winds up being ten times as difficult and ten times more expensive than planning for it in the first place.

Everybody on the team knew this and we tried as hard as we could to work out all the details before we started the shoot. The problem was we didn't have enough time and couldn't change the date of the shoot because we wouldn't have been able to get our two main actors, James Earl Jones and Michael Biehn. Going into the shoot, we had a pretty good idea of how we were going to work out all of the technical details such as camera tracking on a bluescreen, matching lighting to computer graphics, compositing, and so on. However, we ran into difficulties because we didn't allow enough time for the more complex shots and were forced to edit on the fly during the shoot.

An unforeseen problem during the post-production was that we dramatically underestimated the storage and network requirements of working with 60 minutes of digitized video. Westwood has a very robust and fast network with a large amount of storage space, but it was never designed to meet the needs of video post-operation. An amazing effort by the MIS staff and a couple of called-in favors got us enough storage space on the network to keep going.

From the start, the team struggled to get video from digital beta to the SGI- and PC-based compositing systems. Footage was digitized on an Avid system and copied to file servers for distribution to the PCs. The SGIs grabbed the footage directly from tape using built-in digitizing hardware. From the compositing stations, various shots were completed and transferred back to a file server to be compressed and put in the game. This, along with the fact that many individual scenes were worked on by several artists, multiplied the storage requirements several times over. In the end, the video assets were spread across four separate file servers and took up well over 500GB of space.

Not only was space a problem, but moving hundreds of megabytes of files a day from machine to machine became a bottleneck. A few minutes here and there to transfer files doesn't sound like much until you add it all up. If we had it to do over again, we could have alleviated the problem by building a very specialized (and expensive) network, by getting hardware that allowed artists to digitize their footage directly from tape, or by reducing the scope of the project and sidestepping the problem entirely.

4. Locked documents too early. One of the side effects of schedule slippage was that we locked our documents too early in order to achieve the localization plan. We knew this was going to wind up causing us significant pain, but at the time there was nothing we could do to avoid it. The result turned out well, but a lot of time and effort was spent to make everything work together.

At the point when we locked the audio script, mission design and balancing were not complete. As we played through the missions, we realized that certain objectives were not clear and needed to be explained further. The previous method for doing this was to have the in-game AI persona (Eva or Cabal) relay the information to the player through voice cues. This was not an option for TIBERIAN SUN, however, since we made the switch to professional voice talent for Eva and Cabal. Costs and scheduling didn't allow us to do as many pickup recording sessions as we wanted. Also, the locked audio scripts were already localized and recorded, which made recording additional lines out of the question.

The only option available was to redesign the missions or add text to the missions to make the objectives clearer. Redesigning the missions would have added at least a month to the already late schedule, so we quickly ruled that option out. We wound up going with text that popped up in the missions, although the original design called for all text in the game to be accompanied by a voice-over.

5. SCHEDULING PROBLEMS. As with most projects in development today, TIBERIAN SUN suffered from scheduling problems; ours resulted in a nine-month delay. There wasn't a single reason that caused the product to be delayed, but rather a series of seemingly minor contributing factors.

Brett Sperry has a rule of thumb that we often refer to when scheduling projects. When you add one fundamental new technology to a project, it can cause slippage up to 90 days. When you add two fundamental new technologies it can add a year to the anticipated release date. When you add three or more new technologies it becomes impossible to predict the release date of the project accurately.

TIBERIAN SUN features three new systems that resulted in an unpredictable schedule. First, we switched our core graphics engine to an isometric perspective in order to enhance the game's 3D look. This resulted in a cascade effect of broken systems that weren't anticipated. Bridges that could be destroyed and rebuilt, for example, wound up taking over ten times as long to program as we originally estimated. Adding bridges complicated systems such as path-finding, Z-buffering, rendering, unit behavior, and AI.

Scripting was another area in which we added a slew of new functionality. We added an increasing number of triggers to the game to allow the designers flexibility in creating the missions. Each new trigger added was more specific than the last and was used for increasingly rarer conditions. Since triggers could be used in combination, we ended up with an overwhelmingly large number of events that needed to be debugged. We would often fix one trigger to work in a specific situation and inadvertently break the same trigger in a different situation.

AI and unit behavior was the third main area that used new technology. We set out to create a challenging and fun AI that could react to the player's actions and change tactics to compensate. We should have focused on fewer areas of the AI instead of trying to redesign the whole package from the ground up.

Mobile sensor arrays deploy to detect stealth tanks.

Disruptors power up.

Overall Tips

With TIBERIAN SUN, we built the game we originally set out to build over three years ago. Almost all of the new engine features we designed were implemented in the final product, and many more were added along the way. We built a game that is as easy to play as its predecessor while offering up lots of new units featuring interesting tactics. All of this was done while keeping the system requirements low enough to run on most systems: a 166MHz Pentium with 32MB RAM and a 2MB video card.

We learned, or relearned actually, a few more things about making RTS games that weren't listed above. They are:

- If the game has Internet or multiplayer capability, build this functionality as soon as possible since it will let you get into the game and balance it early.
- Don't shield yourself from reality. If your game supports Internet play as well as LAN play, don't play only LAN games and assume that Internet performance is acceptable.
- Keep the story tightly focused on players' actions and don't treat the story as a separate entity. Remember that the player is always the main character.
- Wherever possible, try not to mix disparate technologies (3D visual systems with 2D, for example) that have inherent problems working together. Instead, go back and modify the design.

In the sense that TIBERIAN SUN was a game with lots of expectations for a sequel, it was a lot like *Star Wars: The Phantom Menace*. No matter how the final product turned out, there would be people that complained that it was too much like the original and others who thought it wasn't enough like the original. As a company, we set out to deliver what we intended — a fun new RTS game that offers players a slew of new tactics.

After three years of working on TIBERIAN SUN, it was a great feeling to finally finish the game and see it on the shelves. No matter how many products you ship, that feeling never goes away. TIBERIAN SUN broke Electronic Arts' sales record for the fastest-selling computer game in the 17-year history of the company with more than 1.5 million units sold so far. But best of all, the team is proud of the product they created and can't wait to get started on the next one. ■

Nod laser turrets repel a GDI horde.

Orca fighters escort a transport.

Concept sketch of a GDI Orca bomber.

A Wolverine on the firing range.

by Levelord

Exposing Myself

I am currently working on the PC game version of *Heavy Metal*'s *F.A.K.K. 2* movie. First and foremost on my mind right now is the major hang-up we Americans have with nudity.

As you may know, *Heavy Metal*, both the movie and the magazine, is definitely sexually oriented and has lots of nudity. Yet I'm having a hard time incorporating even the slightest of naughty bits into the game. I'm not advocating sex and pornography in computer games, but I think our culture goes way too far in restricting the exhibition of the unclad physique.

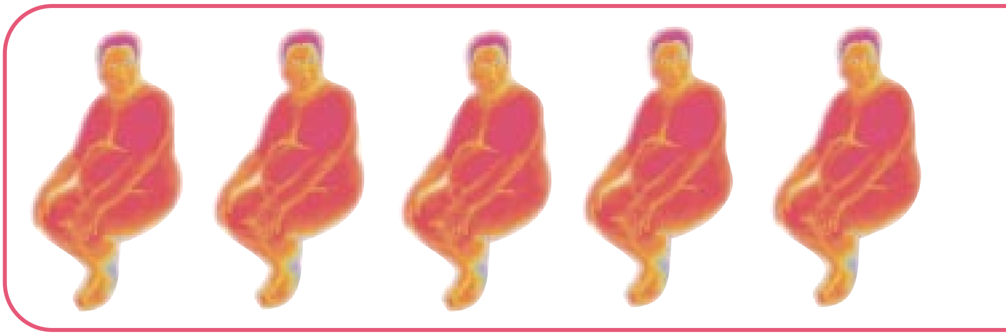
I have been preoccupied with Americans' phobia of nudity a lot lately. I may sound over-reactive and perhaps even juvenile, but after 42 years of living in this hypocritically puritan country, I must jump on the soapbox and protest this prudish disposition of ours. Maybe it's the influence of spending my impressionable years in the 1960s and 1970s among hippies, or maybe *I'm* the one with the true hang-up and doth protest too much, but what I see in our country's attitudes towards the bare human form is diseased and deviant. I'm not saying we should all parade down Main Street in the nude, but as I try to design levels for a *Heavy Metal* game, I am becoming intensely aware of the extreme to which America is anti-body.

Admittedly, any intentions I have to include nudity in a computer game are purely for its suggestive, sexually provocative, and, yes, even gratuitous usage. Are these motives at all unnatural? I remember a report that estimated most people think about sex and nudity at least 12 times a day. Yet as a game developer, I am oppressively banned from even the slightest hint of carnal exposure. I can't show a single nipple in a game without some uptight fundamentalist writing a letter to the store where they bought it and having the store take it off the shelves. Conversely, I can put a baby carriage in the very same game and blow it up with little retail consequence.

The media have been afire the past few years because of all the violence in American entertainment. Although it has been seen as a problem since the early 1960s, it still reigns as one of the underlying mainstays of American amusement. I, admittedly, do find entertainment in violence. I'm not sure if this is an innate behavior due to an evolution through which the most violent aggressor is most likely to succeed, or whether it is indeed simply the influence of this type of entertainment during my formative years. Either way, though, I recognize that it is not correct and the glorification of violence is not a good thing.

Nudity, on the other hand is wholesome and natural. There is simply no denying that, other than perhaps for religious reasons. In our hearts, we may not all be violent, but underneath our clothes, we're all naked. I'm all for ratings systems that classify games and movies so audiences know what to expect, but what I do not understand is why nudity, a completely natural state in which we all arrived in this world, is lumped into the same category as vulgar language and even violence.

By now you must be thinking to yourself, "Hey, this guy just wants to see more naked women!" To which I would be inclined to say, "Yes!" My desire to put nakedness in F.A.K.K. 2 is strictly for the purpose of being sexually suggestive and titillating. Since nudity is natural and occupies so much of our minds anyway, I simply don't see the harm and I don't understand the prohibition.



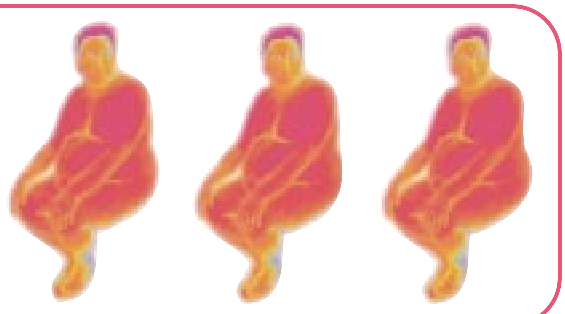
It was my good fortune to have worked on DUKE NUKEM. The two things I hear most about DUKE are its degree of interactivity and its strippers. Not just from immature adolescent teenagers, but from everyone. As an example of our ridiculous national phobia, the original box art showed a stripper with a nipple exposed. A single-pixel nipple, one little brownish square, and we were told that the major retail stores in America would not shelve the game.

Continued on page 59.

The Levelord has been tinkering with games since first downloading the DEU editor for DOOM in 1994. He designed the levels for DUKE NUKEM 3D while at 3D Realms, and the QUAKE add-on pack SCOURGE OF ARMAGON and SIN at Ritual Entertainment, where he is currently working on HEAVY METAL'S F.A.K.K. 2. Contact him at levelord@ritual.com.

Continued from page 60.

What I would most like to see change in our industry — the entire country, actually — is to relax our attitudes about nakedness. Whether an instance of nudity is gratuitous or elegant, I think all Americans need to reexamine their hang-ups.



I have visited Europe a few times and get game-related magazines from many European countries, and it's clear to me that the rest of the world enjoys a far more mature perspective about the naked body (whereas gratuitous violence is deplored). Many foreign cultures allow nudity (which is generally handled in good taste) to be seen by even the youngest of audiences. Anyone who's ever visited a beach on the Riviera knows that Europeans don't possess the knee-jerk aversions to nudity that Americans have. Even their television and billboard ads show nudity with the only connection between the merchandise and the naughty bits being the fact that they recognize and celebrate the allure of the naked body.

Nudity is a natural condition that will not disappear anytime soon, and I'm constantly reminded of how frustrating our own farcical prohibition really is. ■