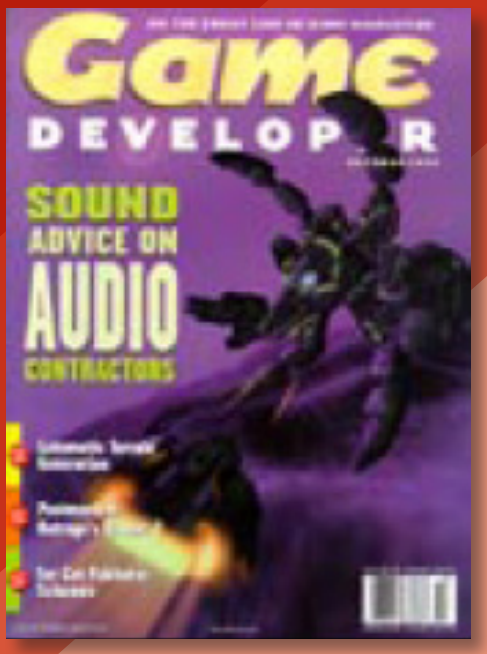
gd

GAME DEVELOPER MAGAZINE

OCTOBER 1999



UBM Tech

Graphics Fly... Will Developers Fry?

This heard at a Siggraph panel: "Consumer graphics cards in two years will be more powerful than any graphics card available today at any price." That's quite a bold prediction, but I agree. Graphics hardware has entered a phenomenal technological growth spurt, thanks in part to the demands of today's games. The latest crop of consumer 3D chips, such as Nvidia's GeForce 256 (formerly known as NV10), boasts features that were found exclusively on high-end workstation cards only a year ago. Consequently, the line dividing "consumer" and "workstation" class cards gets fuzzier each passing day.

The scary aspect to the aforementioned prediction is that, if it pans out, we've got a lot of work ahead of us if we hope to maximize the capabilities of that hardware. Or, to put it another way, where are those hundreds of millions of triangles being drawn on screen each second going to come from? It's apparent that game developers are increasingly playing catch-up with hardware. Without changing the way things currently work, we risk widening the gap between what hardware (both PC and console) supports and what games actually deliver. To narrow that gap, several things need to happen.

First, we need more specialized tools. Many SDKs currently categorized as "middleware," such as physics and 3D animation engines, are often expensive to license and don't appear to integrate easily into already-underway projects. Perhaps this will change over time as more commercial SDKs hit the market, causing prices to fall and developers to become better acquainted with using comprehensive off-the-shelf libraries for adding complex functionality into games. But the types of products that I think will make the most impact on game development will be highly focused — those that sacrifice broad feature sets in favor of super-specific functionality (for example, a tracking camera system for TOMB RAIDER clones, or a flexible terrain editor/generator) and reduced price (what I call "Smacker pricing" — $5,000 or less per product, no royalties). These will be inexpensive tools that even junior developers can learn in a few weeks, and which can be integrated into a game quickly. Where will we get these dream tools? That brings me to my second point.

At Siggraph, it was evident that the graphics research community desires closer ties to the game development industry. The problem is, researchers don't know how to build those relationships with us, how to identify what aspects of their research we might find useful, and how to craft licensing schemes we'd find attractive. So in order for our commercial interests to be served, it looks like it's up to us to build those relationships. How many people on your team regularly investigate technologies that are still in the labs? It might be worth your while to pick up the Siggraph proceedings each year, see what's been presented, and make licensing offers to the appropriate researchers when bits of technology look useful. Bridging the gap between the public and private sector could give developers access to reasonably priced, highly specialized software tools for maximizing the horsepower of graphics hardware.

Finally, we can't be afraid to take risks with technology. Gambling on the capabilities of future graphics hardware seems to be one of the safest decisions you can make today. In this month's Postmortem of DESCENT 3, for instance, Jason Leighton and Craig Derrick discuss the reluctance they felt when early in the project they decided to build a graphics engine that *required* 3D hardware acceleration. Of course, any AAA game aiming for the high end of the market today takes 3D acceleration for granted, but surely there are technologies today which look just as risky as hardware-accelerated 3D did in back 1996. With better tools and market foresight, the likelihood of a long death march to get your game out will be reduced substantially. ■

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
Cynthia A. Blair *cblair@mfi.com*

EDITORIAL

Editorial Director
Alex Dunne *adunne@sirius.com*

Managing Editor
Kimberley Van Hooser *kvanhoos@sirius.com*

Departments Editor
Jennifer Olsen *jolsen@sirius.com*

Art Director
Laura Pool *lpool@mfi.com*

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

Contributing Editors
Jeff Lander *jeffl@darwin3d.com*
Paul Steed *psteed@idsoftware.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 Harmonix
Dan Teven Teven Consulting
Rob Wyatt DreamWorks Interactive

ADVERTISING SALES

Western Regional Sales Manager
Jennifer Orvik e: *jorvik@mfi.com* t: 415.905.2156

Eastern Regional Sales Manager/Recruitment
Ayrien Houchin e: *ahouchin@mfi.com* t: 415.905.2788

International Sales Representative
Breakout Marketing e: *breakout_mktg@compuserve.com*
t: +49 431 801703 f:+49 431 801797

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 Director Gabe Zichermann
MarCom Manager Susan McDonald
Junior MarCom Project Manager Beena Jacob

CIRCULATION

Vice President/Circulation Jerry M. Okabe
Assistant Circulation Director Sara DeCarlo
Circulation Manager Stephanie Blake
Assistant Circulation Manager Craig Diamantine
Circulation Assistant Kausha Jackson-Crain
Newsstand Analyst Joyce Gorsuch

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
CFO Ed Pinedo
Executive Vice Presidents Darrell Denny, Galen A. Poss, Regina Starr Ridley
Sr. Vice Presidents Annie Feldman, Howard I. Hauben, Wini D. Ragus, John Pearson, Andrew A. Mickus
Sr. Vice President/Development Solutions Group KoAnn Vikören
Group President/Division SF1 Regina Ridley

BPA INTERNATIONAL

www.gdmag.com

Playing the Political Game

Reading Herr Kreimeier's chilling vision of game censorship in Germany ("Killing Games: Violence vs. Censorship," Soapbox, August 1999) has forced me to briefly divert my attention from developing games to worrying about the future of the game industry. The thought that in a few years I might not be able to develop or play games where things "blow up good" is not one that I am happy about. I have no problem with rating systems, but the suppression and confiscation of games based on their content reminds me too much of book burning.

When I've taken time to consider the issue (rarely), I've always thought of the "anti-violence in videogames" politicians as grand-standers using their nutty ideas to win votes from Ma and Pa Bible Belt. I always figured they'd have zero chance of actually getting anything passed. However, knowing what has been done in Germany and seeing the rise to dominance of mass media publishers who would probably just roll over and accept any such changes, I now believe that I should involve myself in this issue.

Unfortunately, like most of my colleagues, I have only a very small amount of time to devote to a crusade. My game's got to ship by Christmas, right? I vaguely remember a game lobby group that was formed a few years ago to combat this type of censorship, and I plan to get involved in such a group.

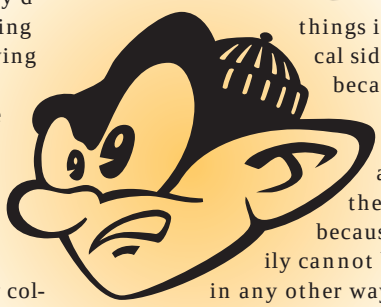
I'd like to thank *Game Developer* and Bernd Kreimeier for bringing the possible outcome of goofy anti-videogame legislation to my attention. They'll take away my fighting games when they pry the game-pad from my cold dead hands.

Lee Waggoner, President/CEO
Wrecking Ball Software Inc.
via e-mail

Get out of the Conference Room

I don't agree with Doug Church's proposal that our industry needs a lexicon in order for design to go forward ("Formal Abstract Design Tools," August 1999). Languages are not created by committee, they are formed out of necessity. The movie term "zoom-in dolly-back" was not created because a bunch of directors decided that they needed to create terms for all the different shots they did. More likely than not, some director said "O.K., in this scene I want the camera to zoom in while we move the dolly away from the scene," and when they shot the scene he probably said, "O.K., now zoom in and move the dolly back!" and of course the phrase got shorter and eventually caught on with other directors — this is how language develops. The reason there are so many terms for things in the technical side of games is because they are needed. It is formed along with the technology because it necessarily cannot be described in any other way.

We'll lend you an ear. E-mail us at saysyou@gdmag.com. Or write to *Game Developer*, 600 Harrison Street, San Francisco, CA 94107.



Not only can we not create a lexicon via committee, I also believe that it is not necessary. Any designer who's worth what he's paid will be able to sit down, play a game, and then tell you why he's still playing it 20 hours later. If you can't figure out what makes a game fun or addicting then you are probably doomed to making boring games unless you just get lucky. I don't think this industry is run by a bunch of lucky designers — I think most of us know what we are doing. Design does not evolve in the same way as technology because it is abstract; it is an art. Technology evolves as fast as we can push it, but you cannot push art. It will evolve at its own pace and all attempts to push it ahead of its time will most likely end in failure.

Designing games is an art, and art cannot be defined so easily by a few jargon-like terms — that is for technology and other less abstract concepts. You cannot force jargon into existence — it is formed out of necessity, and I for one do not believe there to be a necessity here as of yet. Art evolves in canvas, not in conference rooms.

Adam Heine
via e-mail

AUTHOR DOUG CHURCH RESPONDS:

"Lexicon" is Game Developer*'s word, "Jargon" is your word. "Tools" was my word.*

The underlying tool idea and specific tools discussed in the article have evolved through use. During development of UNDERWORLD *1 and 2, we talked about our goal to "immerse" the player. On* SYSTEM SHOCK *1, we discussed more seriously what tools we had to work with. As I stated in my talk on Intention at the 1997 CGDC,* SHOCK *had no conversation mode because we felt that UW conversations had little player intention, especially compared to the 3D world. This isn't a judgement of conversation, but rather a statement about what tools fit our game design vision. Tools are used to shape games. We continue to try and understand the tools we use and how they work.*

As I see it, "usage" has been happening for years, the GDC talks and this article are "exposure," and we'll see if the tools idea catches on. Step 2, right? I didn't sit around in a conference room trying to figure out a topic for a Game Developer *article. I'm a game designer trying to share ideas several teams have actually used in attempting to implement a game play vision.*

I strongly disagree that design is a mysterious, unanalyzable "art." Of course vision and aesthetics are involved, but they are part of technology or books, too. Stating design is not rationally understandable, unlike those other "simple" things, feels lazy to me. As a game designer, I should learn to design games, right?

Some people have interpreted my advocacy of identification, analysis, and understanding of design tools as an attack on designers. That was not my point. Nor was I saying people who don't use big words are losers, pedantic analysis is game design, or that tools encapsulate all. Certainly, you can dismiss design analysis as pointless or attack it as wrong. However, I believe the tools idea and the process of developing it has helped us focus game play and better achieve our vision in titles I worked on. Therefore, refinement of that understanding will continue to bring benefit. That is the spirit in which the article was written.

BIT Blasts

News from the World of Game Development

New Products

by Jennifer Olsen

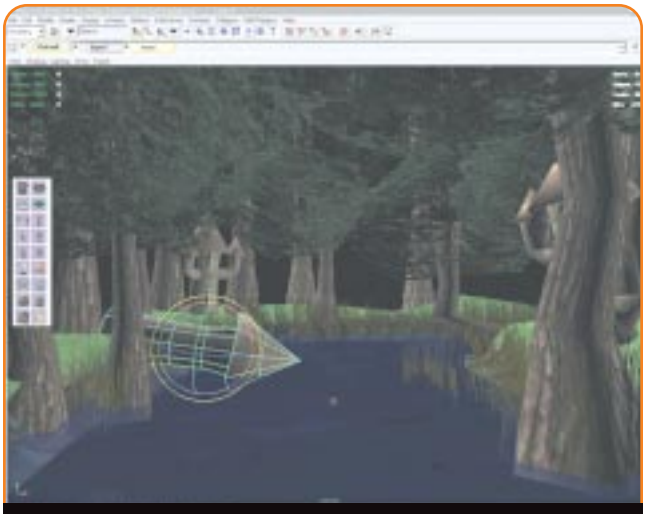
If You Build It, Will They Come?

ALIAS|WAVEFRONT is still busy wooing game developers, and their latest serenade is in the form of Maya Builder, designed to streamline level design for game designers and 3D programmers, while allowing easy integration with other Maya features they're already using.

Builder includes features from Maya's suite of polygonal tools, including Artisan, prelighting, UV editing, and nonlinear deformers. Its basic animation capabilities, which can animate objects such as drawbridges or cranes, support Maya's standard keyframe-style tools and IK solvers (with source code, so programmers can deploy behaviors easily on their end platforms). Maya's Embedded Language (MEL) lets programmers customize Builder with level editing tools specific to their engines, while the Maya API offers access to Maya's internal data structures for creating custom translators and plug-ins.

Maya Builder will be available for IRIX and Windows NT, and is expected to have a retail price of $2,995. It will be included in the upcoming release of Maya 2.5 later this fall.

■ **Alias|Wavefront**
Toronto, Ontario, Canada
(416) 362-9181
http://www.aliaswavefront.com



Maya Builder will be one of the new features of Maya 2.5, offering new tools specifically for level design.

Riding the Next Big 'Wave

NEWTEK unveiled Lightwave 6, the next generation of its modeling and animation software, at Siggraph this past August. Newtek claims this particular release is their most significant upgrade of the past ten years. No longer content with red-headed-stepchild status among character animation systems, the new Lightwave offers several enhancements tailored specifically to game developers.

Version 6 introduces a family of character animation tools called Intelligenties, which consist of Skelegons, Endomorphs, and Multi-meshes. Skelegons are polygons that appear like regular 3D bones, so that any changes made to the character automatically update the skeleton as well. Endomorphs are designed to help with complex morphing tasks (such as facial animation), allowing changes of expression, mood, or actions by training a single model. Multi-meshes, as the name implies, embed multiple layers of geometry into a single object, aiding in the management of complex hierarchies.

Also notable are the new rendering technologies offered (including souped-up Hypervoxels) and a new hybrid Inverse/Forward Kinematics engine. In addition, Newtek has overhauled its curve editor, offering multiple-tangent types such as Bézier and Hermite curves, which it hopes will improve control of motion curves, resulting in better realism in character animations.

Lightwave 6 runs on Macintosh, Alpha, IRIX, and Windows NT systems, and carries a suggested retail price of $2,495. Registered 5.6 users can upgrade for $495.

■ **Newtek**
San Antonio, Tex.
(210) 370-8000
http://www.newtek.com

Not Your Father's 3D API

GLOBAL MAJIC SOFTWARE recently launched 3DLinx, a new 3D development environment for Windows incorporating a variety of languages, including Visual Basic, Delphi, C++ Builder, and others.

Traditional 3D APIs can have long learning curves, depending on the skills and experience of the programmers using them. Global Majic has attempted to simplify many complex tasks such as scene graph management, view culling, and collision detection, to offer 3D programming to a wider audience.

The Standard Edition of 3DLinx, priced at $895, is geared toward quick creation of 3D applications, while the Professional Edition, for $2,495, includes additional loaders for importing different file formats. An SDK will also be available for third-party development of custom features and add-ons.

■ **Global Majic Software**
Huntsville, Ala.
(256) 922-0222
http://www.3dlinx.com

Industry Watch

by Alex Dunne and Dan Huebner

NOT SEEN ON THE SCENE. At Siggraph, the usual suspects such as SGI, Discreet, Alias|Wavefront, and others had large booths on the floor, but two companies were conspicuously absent from the exhibition: Microsoft and Apple. Rumor has it that Apple won't attend tradeshows where it can't have the largest booth, so perhaps its excuse was that it couldn't compete with SGI's dueling 60×60-foot booths at the show entrance. But what was Microsoft's excuse? Microsoft Research, which employs some of the most respected graphics experts in the industry (Glassner, Blinn, Hoppe, Whitted, and on and on), had a number of people speaking at the conference, but one wonders why the biggest computer graphics event of the year wasn't on the company's radar.

STOLAR AND SEGA PART WAYS. Bernie has left the house. The question is, why? As the president and COO of Sega of America, Bernie Stolar was instrumental in shaping the Dreamcast launch, but Sega hasn't revealed any details about the split. As you might imagine, morale around SOA wasn't terribly high when it was announced, but it's hoped that Toshiro Kesuka — Stolar's replacement as vice-chairman and COO — will keep the ship afloat through this critical time for the company.

Bernie Stolar helped shape the U.S. Dreamcast launch.

VOULEZ VOUS MIDWAY GAMES? Midway regained the rights to distribute its games in international markets after reaching a settlement with GT Interactive. Midway, which had filed a securities lawsuit against GT Interactive, must pay GT an undisclosed sum, and will distribute its games internationally from now on. Midway can now directly realize the revenues and profits from such worldwide sales, which wasn't the case under its previous agreement with GT. Under the terms of the terminated distribution arrangement with GT Interactive, Midway did not participate in the revenue and profits generated by sales of its home videogames in international territories until certain nonrefundable advances were recouped by GT Interactive.

If titles such as Ready 2 Rumble are a hit overseas, Midway won't have to share the prize money with anyone.

ESRB O.K.'S PERMISSION SLIPS. Some game web sites now have something in common with elementary school field trips: parental permission slips. The Entertainment Software Ratings Board granted its first certification in a program that requires children under the age of 13 to have a "permission slip" signed by their parents before they will be allowed to participate in any web-based activities. Electronic Arts is the first company to implement the system, called ESRB Privacy Online. Youths aged 13 to 17 registering at an Electronic Arts site will have a notice of their registration mailed to a parent, who then has the option of removing their child's name. Those younger than 13 will be directed to print a permission slip to be signed by a parent and returned to EA. The program is not designed to protect children from objectionable content online, but rather is geared towards regulating the collection, use, and safeguarding of customer information.

TAKING ON VIRTUAL SWASHBUCKLERS. The Interactive Digital Software Association (IDSA), along with six member companies, filed a lawsuit in Northern California against six online pirates located within the United States. The IDSA alleges that the defendants posted pre-production copies of PC and videogames on the Internet. Named as defendants were the leaders of the Class, Paradigm, and Razor 1911 hacking groups, operating in San Francisco, Dallas, Austin, Minneapolis, the Philadelphia area, and the Champaign, Illinois, area. Acclaim, Accolade/Infogrames, Bethesda Softworks, Interplay, LucasArts, and 3DO are also plaintiffs in the case. The lawsuit charges that the defendants engaged in copyright and trademark piracy, racketeering (including mail fraud, wire fraud, and interstate transportation of stolen property), counterfeiting, and unfair competition. The IDSA gathered evidence in the case by monitoring the groups for several months, tracking their private chat sessions and message postings. It's the first case ever filed against actual named members of any piracy group. ■

UPCOMING EVENTS CALENDAR

Digital Video Conference and Expo

LONG BEACH CONVENTION CENTER
Long Beach, Calif.
October 19–22, 1999
Cost: variable
http://www.dvexpo.com

Game Developers Conference 1999 RoadTrips

OGDEN ECCLES CONFERENCE CENTER
Salt Lake City, Utah
November 1, 1999

THOMPSON CONFERENCE CENTER AT THE UNIVERSITY OF TEXAS
Austin, Tex.
November 3, 1999

Cost: $120 ea. (discounts available)
http://roadtrips.gdconf.com

NDL's NetImmerse 2.3

by Jonathan Blow

NetImmerse is a software development kit for 3D graphics and animation. It is a scene graph API, meaning that objects in your 3D world are stored in a hierarchical tree. The graphics engine then renders your scene by traversing the tree.

You make things happen in the world by tweaking the attributes of objects in the tree. If you transform an object (rotate it, move it, or scale it), all of its children (nodes below that object in the tree) will be transformed as well. To make an object move through space, you can either modify its position manually from frame to frame, or you can attach animation curves to its attributes; the object will follow the path of motion described by the curves.

The nodes in the tree can be simple geometry objects such as textured triangle meshes; they can be more complex geometry objects, like curved objects made from Bézier patches or height-mapped landscapes; or, they can be completely new objects that you define yourself.

NetImmerse's name can be misleading; it sounds as if it's a networking API, but it's not. There are a few features of NetImmerse that can be used in conjunction with networking (such as the ability to asynchronously load geometry and animations into a running scene, or fetching an image file from a given URL), but this is nowhere near its central point.

NetImmerse provides components that include the following technical features:

- A terrain rendering system with view-dependent detail reduction, based on Peter Lindstrom's algorithm from the 1996 Siggraph proceedings. (This algorithm has been popular with game developers creating their own terrain systems.)
- View-independent detail reduction for triangle meshes. This appears very similar to Hugues Hoppe's progressive mesh algorithm, which is included with DirectX. I find that NetImmerse's version is nicer and easier to use.
- A portal system, for culling hidden geometry in indoor environments.
- A collision detection system that uses trees of oriented bounding boxes (the OBB-Tree algorithm developed by Gottschalk) and some other simple bounding primitives such as spheres, extruded spheres, and extruded ellipses.
- A system for dynamic tessellation of curved objects built from Bézier patches.
- A shape animation system that includes morphing, skinning, general path control, and particle system support.
- A multitexturing texture system with a cache manager.
- A 3D sound API that uses Aureal Semiconductor's A3D 2.0 as its back end.
- A rendering manager that can use Glide, Direct3D, or OpenGL as its back end.

QUALITY OF TECHNOLOGIES. So NetImmerse provides all these features. But how good are they, actually? The answer is, pretty darn good for a generalized API.

The source code for the NetImmerse internals is clean. Each of the modules is a competent implementation of the targeted technology, written to be as generally useful as possible while still being efficient and easy to modify.

However, none of the modules is as good as what you'd get if you told an experienced 3D programmer to create that functionality for one specific game. For example, the NetImmerse terrain module does not provide smoothly interpolating terrain vertices (you see "popping" effects) and is not structured for landscapes that have very high texture densities. It is also not very memory efficient compared to what you could create if you knew exactly what terrain size and resolution you wanted and implemented the Lindstrom algorithm by hand. So if you are developing a game to compete with STARSIEGE TRIBES 2, and you want your terrain to be superior, you're going to have to write your own.

Similar statements can be made about every component of NetImmerse. Though the collision detection systems are quite intelligently coded with a lot of care given to speeding up the special cases between different types of primitives, they don't report accurate enough collision points or times to be used for "real" physical simulation. The portal system doesn't perform optimally for some geometry types, which may be the ones you want to use in your game. And so on.

These observations can't really be taken as criticisms of NetImmerse, however. These are the same problems that come up any time you use a generalized system to perform a specific task. Because the system is attempting to solve a harder problem than what you're using it for, it will not be as efficient as something that is tailored to your specific needs.

NetImmerse is designed for modularity. So if you need a terrain system that is more memory-efficient and that smoothly interpolates vertices, you can write it yourself and stick it into the scene graph, and it will work. But if you're pushing the cutting edge of performance, there's a limit to how effective this will be; at some point you have to re-architect the game engine around your core features, and this would require throwing away most of the structure that NetImmerse gives you, or using it in an ancillary way.

ASSET CREATION. NetImmerse does not provide a content creation tool for 3D geometry and animation; the paradigm is that you should use the commercial authoring tool that works best for you, then export the data for use with NetImmerse. NetImmerse provides exporters for 3D Studio Max 2.x, Softimage 3.8, Multigen Creator, and

Jonathan Blow is the guy who receives e-mail sent to jon@bolt-action.com.

Motion Factory's Motivate.

NetImmerse can load textures from .TGA or .BMP files and from files consisting directly of RGB 3-byte values or RGBA 4-byte values. It can also parse textures out of its own file format, .NIF. The NetImmerse file format is a package format that can contain any of the NetImmerse data types; you get one of these when you export a scene, or when you use API calls to save a scene that you have created in memory.

DOCUMENTATION AND SAMPLE CODE. The system is fully documented and the documents are provided in two file formats: .PDF and Word 97 .DOC. The documentation is of high quality; in fact, it's the best documentation I have ever seen for a game-related SDK, by a wide margin. The first piece of documentation is a 26-page programming manual that provides an overview of the entire system; after reading it, anyone who has ever used a scene graph API will be able to sit down and begin coding. Anyone who has not will want to proceed to the next piece of documentation, the tutorial, which is a walkthrough of every sample program provided with the SDK.

There are 15 different sample programs, and the documentation does a terrific job of explaining them. Think of a good programming book that you would buy for $70 in a bookstore, wherein the author creates programs and guides you through their logic one section at a time. That's what you get in the tutorial (all 116 pages of it).

If you're in a rush and want to skip the tutorial, that's O.K. too. The sample source code is clean and easy to read on its own. There is not a trace of platform-specific code in the samples, which is nice for two reasons. First, it shows that the underlying architecture is clean and complete, without falling back on Windows for some functionality or allowing Windows data structures to infiltrate the API calls. Second, it makes the code pleasant and easy to understand (because it's not filled with the nasty type definitions and unreadable naming conventions common in Microsoft APIs).

FIGURE 1. *This shows NetImmerse's continuous LOD demo. As you move the dinosaur into the distance, its polygon count decreases. Popping is scarcely visible.*

The rest of the documentation describes NetImmerse's major technological components (the portal system, detail reduction system, and so on). These files use diagrams to describe the operation of each component; they also give guidelines on the optimal, proper, and improper uses of each component.

HOW IT STACKS UP AGAINST COMPETITION. Anyone who is considering licensing a toolkit like NetImmerse will, naturally, consider other options as well. Of course one can license a tool that competes with NetImmerse and provides functionality in roughly the same arena; since this is not a review of the whole marketplace, we won't touch on those here. However, we will discuss two important alternatives, in terms of their scale of functionality, to put NetImmerse into perspective.

FIGURE 2. *NetImmerse offers projected shadow targets, shown, as well as regular ground-plane projection, in which case the shadow would disappear under the wall.*

First, you might opt to license the engine from an existing 3D game that you like, such as UNREAL or QUAKE 3. On the minus side, this option will cost you a lot more than a tool like NetImmerse. However, what you get with, say, the QUAKE 3 engine, is an already-assembled game that you can modify until it becomes what you want. If the original game is technologically similar to the game you want to develop, then you won't need to spend too much engineering effort to get your game into a basic running state. Instead, you can spend all your effort on refining your game, which hopefully means it is of higher quality in the end. NetImmerse provides a lot of features, but it is not a game or a game engine. You will still need to do a lot of work to put the pieces together in the right way, and to provide the pieces of core functionality needed to make your game run.

A second alternative might be to use Fahrenheit, the codename for the upcoming version of Direct3D, which reportedly will contain its own scene graph API. (Fahrenheit specifications are not yet available for public perusal.) The structure of Fahrenheit should be similar to NetImmerse in many ways, and Fahrenheit will most likely be free for general use, as with previous versions of Direct3D. So why would you opt to use NetImmerse, which will cost you? First, Direct3D has a history of not living up to its promises, so expect Fahrenheit to disappoint and/or annoy programmers for a while after its initial release. Also, it is quite unlikely that Fahrenheit will be ported to any non-Windows operating systems, so if your game relies on it, you will be tied to Windows. And you can bet that you won't get Fahrenheit source code, so NetImmerse will be easier to adapt to fit your needs. Finally, the

authors of NetImmerse seem to be more in touch with the needs of game programmers than the designers of Fahrenheit. Unfortunately, space is not available for a full discussion of this. Simply stated, all of NetImmerse's features can save game developers significant amounts of engineering time, while Fahrenheit's features seem to have been motivated by a checklist of buzzwords. It is likely that game developers will ignore many of Fahrenheit's features, seeing them as ill-fitting or irrelevant. But a price tag of "free" goes a long way, so we can be sure that Fahrenheit will be used.

LICENSING TERMS. NetImmerse can be modularly licensed. The most basic licensing of the SDK, for use on a single game project, costs $50,000. The portal, terrain, and mesh detail reduction systems cost extra, at $10,000 each. The MacOS version of the SDK (which NDL plans to make available in fall 1999) also costs $10,000. This is a royalty-free, no-strings-attached licensing agreement. For double these prices, you can license NetImmerse for an unlimited number of games to be developed at one site.

The import/export tools are available in binary form free of charge. The source code can be licensed for $20,000 per tool. Extended maintenance and support licenses are also available.

BUILD OR BUY? It would cost many times NetImmerse's price tag to develop the in-house equivalent of NetImmerse's code base, and it would also take a lot of time (measured in years). So in terms of cost and time-to-market savings, licensing NetImmerse makes sense, if your goals are compatible with the usage of a general-purpose API.

We're all familiar with the schedule bloat and general increase of project failure that has developed since games went 3D. From that standpoint, licensing NetImmerse provides a certain amount of security. You know that you have in your hand algorithms that solve a certain set of problems, so a fair degree of the technological risk factor is eliminated from the development process.

Unfortunately, when faced with the choice of "build or buy?" game development teams have chosen "build" more often than they should, resulting in schedule overruns and canceled games. I have seen many popular games on store shelves that could have been developed more quickly, for less money, and been less buggy in the end, if tools like NetImmerse had been used.

The truth is that the vast majority of modern 3D games do not push the limits of the computer's graphics capabilities, even when the game developers begin the project with the intent of doing just that. The developers find that the project was much more difficult than they expected, and despite the schedule overruns they still don't have time to optimize their hand-coded system to be as fast as it could. In the end, the game player needs a very fast computer to play the game, not because the game has a very sophisticated graphics engine, but because that engine is not sophisticated enough. Thus one of the major reasons behind the "build" decision — to have a hand-tuned system that is superior to what everyone else has — becomes self-defeating, and "buy" would have been a better choice. ■

NetImmerse 2.3: ★★★★★

Numerical Design Ltd.
Chapel Hill, N.C.
(919) 929-2917
http://www.ndl.com

System Requirements:
For development environment: 200MHz PC, 32MB RAM, 150MB disk space, Windows 95/98/2000/NT, DirectX 6.1 or OpenGL 1.1.

Supported Platforms:
Windows currently; MacOS and Playstation 2 versions are pending.

Pros:
1. Provides components that are relevant to modern games.
2. Source code is clean and accessible.
3. Runs at a reasonable speed.

Cons:
1. Suffers the usual problems of a generalized API — it's generalized.
2. Pricing is out of range for hobbyists.
3. In some cases, an engine from a completed game may be a more effective starting point.

Over My Dead, Polygonal Body

Amazing things have been created with computer graphics over the past several years. Visual effects companies are poised to tackle one of the most difficult challenges in computer-generated (CG) imaging: creating a realistic CG human character. However, in my opinion, it hasn't happened yet, no matter how many times I see life-like organic aliens and robotic bipeds trotting all over the screen.

The trouble is, humans are tough to simulate, regardless of how long we take to create the image. It may be obvious, but the difficulty lies in the fact that we are all very familiar with how humans look. We see them all the time. In the morning, I see something resembling a human in the mirror while I shave. Living as I do along the coast, I often see more of the human form than is probably healthy both for my ego and my digestion. In everyday life, we see all varieties of people performing every imaginable action. Each of us is an expert in determining the believability of a CG human form. If the skin looks wrong or the motion looks stiff or the lip-synch is off, each of us screams, "Fake!"

But these technical and artistic problems are solvable. The brilliant artists and technicians charged with making us believe will make sure that it happens. I have thought a bit about the problems this will present, however. I think about John Wayne. What if, while he was alive, he was asked to do a commercial for beer? Perhaps his answer would have been, "Over my dead body!" That used to mean something, that there was no way you would ever get me to do something. A bit of stock video footage and some clever video processing tricks and voilà — John Wayne's dead body selling beer.

Now don't get me wrong. I see nothing wrong with a family member licensing the likeness of a relative for commercial or charitable purposes. I think most people would be glad to continue to provide a living for their family even after they have gone from this life. I just think technology has forced us to reexamine that particular ultimatum. Perhaps it is time for something like, "Over my dead body and virtually extinct telepresence." Kind of loses that Wild West charm, doesn't it?

Bringing Them Back in Real Time

Concerns over these kinds of legal dilemmas are not going to stop me from doing my job, however. As a creator of real-time 3D graphics, I not only face the hurdles which confront my visual effects comrades, but I also need to make these realistic characters move fast. Fortunately, in this task, technology is on my side.

Realistic characters require lots of polygons to make them look realistic. Lots of polygons mean lots of vertices being transformed by an overwhelmed CPU. At Siggraph 1998, I began to see the emergence of transformation acceleration in the consumer graphics hardware space (see "Taking a Break for Siggraph," Graphic Content, October 1998). At that time, I thought it wouldn't be long before we would be able to take advantage of hardware acceleration for transformation and lighting (HW T&L) in our games. Well, Siggraph has come and gone again and we are starting to see this become a reality.

One consumer graphics chip company, Nvidia, has publicly committed to delivery of hardware transformation and lighting with their next generation of graphics chips. There are rumblings that others are likely to deliver on this promise as well. In fact, Microsoft is so certain that hardware T&L will be a reality in the consumer hardware market, they have included support for it in the next version of the DirectX game programming API, DirectX 7. Game developers who are scheduling projects for release in the next production cycle need to consider how their projects will handle hardware T&L.

How Do We Deal with HW T&L?

Fortunately, the driver writers will do most of the work for us. Games using the transformation and lighting pipeline in OpenGL will take immediate advantage of the hardware if it's available. Now, with the introduction of DirectX 7, games supporting this API will also transparently benefit from the new hardware. By using the built-in transformation pipeline in either API, games will get faster.

This will naturally enable games to increase polygon counts without sacrificing performance. It also means that the load on the CPU will decrease,

When not being frightened by the idea of virtual versions of himself, Jeff creates 3D graphics for Darwin 3D. Let him know he is just being paranoid by e-mailing him at jeffl@darwin3d.com.

FIGURE 1A. *Modeling organic objects is a challenge for 3D programmers dealing with hardware T&L.*

FIGURE 1B. *Skeletal deformation systems offer both flexibility and good looks.*

allowing more processor time for game play features, such as artificial intelligence and game physics. And, as to be expected, creating content that performs well on a variety of user system configurations will be more important than ever.

LISTING 1. *Code for the `World->RestBone` matrix.*

```
GLvoid COGLView::GetBaseSkeletonMat(t_Bone *rootBone)
{
/// Local Variables ///////////////////////////////////////////////////////
    int loop;
    t_Bone *curBone;
    tMatrix tempMatrix;
///////////////////////////////////////////////////////////////////////////
    curBone = rootBone->children;
    for (loop = 0; loop < rootBone->childCnt; loop++)
    {
        glPushMatrix();

        glTranslatef(curBone->b_trans.x, curBone->b_trans.y, curBone->b_trans.z);

        // Set observer´s orientation and position
        glRotatef(curBone->b_rot.z, 0.0f, 0.0f, 1.0f);
        glRotatef(curBone->b_rot.y, 0.0f, 1.0f, 0.0f);
        glRotatef(curBone->b_rot.x, 1.0f, 0.0f, 0.0f);

        // Grab the Matrix that is built up to this point
        glGetFloatv(GL_MODELVIEW_MATRIX,tempMatrix.m);
        // Invert this matrix to get the Base->World matrix
        InvertMatrix(tempMatrix.m,curBone->baseToWorldMat.m);

        // Recursive call if the bone has children
        if (curBone->childCnt > 0)
            GetBaseSkeletonMat(curBone);

        glPopMatrix();

        curBone++;
    }
}
```

Since we the graphics programmers can gain this benefit without doing any real work, hardware T&L doesn't seem to affect us much. The programmer will need to conform to a standard pipeline. For the transformation, this doesn't seem to be a big deal. Most 3D programmers are comfortable with matrix manipulation of vertices. The problem is pretty well solved and the only issue of trust revolves around the quality of the driver implementation. Once the hardware is doing the work, the driver issue largely goes away. Most will gladly accept this pathway to hardware nirvana.

Lighting, however, is a much more contentious issue. No one I have talked to is satisfied with the lighting model provided by the OpenGL and DirectX libraries. This issue doesn't bug me that much. I don't know anyone who is ready to give up lighting tricks such as shadow maps and texture-mapped lighting for simplistic Gouraud lighting. While we do need a hardware solution for realistic lighting, this is not it. I advise continued reliance on those lighting tricks. However, as a supplement to pre-computed lighting for dynamic changes or shadow computation, a few hardware point and spot lights couldn't hurt, particularly if they're fast. I can think of a lot of things that a few lights could be used for.

No, simple transformation and lighting is not the problem with hardware T&L. We run into the real trouble when we consider what it means for modeling organic objects.

T&L for Non-Rigid Bodies

Hardware transformation is ideally suited to the display of rigid objects. You first set up the transformation matrix and then submit an object to be drawn. With hardware, it will be costly to obtain the results of the transformation

before the object is drawn. This means everything needs to be set up beforehand.

That's fine for most objects and environments. However, characters do not look their best when composed of rigid objects. As I have explored in a previous article ("Skin Them Bones," Graphic Content, May 1998), creating a character from a single mesh and then deforming it via a skeletal system provides a better solution. By using skeletal deformation, you maintain the good looks of a seamless mesh and the animation flexibility of a hierarchical character system.

Unfortunately, this system requires manipulation of vertex coordinates. Let me review how a skeletal deformation technique works. I have a character in a rest pose in Figure 1a and have created a skeleton for the character in Figure 1b.

In order to deform the mesh with the skeleton, I need to assign each vertex to a bone or set of bones. For example, all of the vertices in the head region should be assigned to the head bone. For now, let me assume that the vertices in the character's head are completely, or 100 percent, assigned to the head bone.

I can determine the position and orientation of the head bone at the rest frame by creating a matrix that represents the transformation needed to move that bone from the origin to its location in the hierarchy. The matrix of each bone is dependent on the matrices of all of its parents, so it is necessary to traverse the entire hierarchy to determine this `World->RestBone` matrix.

Now this matrix will take a vertex and transform it to the location of the bone. However, when I am taking the rest position of the character, I will need to know how to take a vertex in the mesh and transform it back to the origin. Fortunately, since we are using a matrix operation, this is a simple matter of inverting the `World->RestBone` matrix with a standard 4×4 matrix inversion routine. I now have a `RestBone->World` matrix. This only needs to be done once, so I can store this matrix for later use. You can see the code for computing the `RestBone->World` matrix in Listing 1.

I now have the matrix I need to move any vertex from the rest position back to the origin. Now I want to move the vertex to its final pose as shown in Figures 2a and 2b. I can do this virtually the same way. I go through the hierarchy and create a matrix for each bone that will take a vertex from the origin and move it to the bone position in that animation frame. For this `World->Bone` matrix, I don't need to invert it.

I now have everything I need to take a rest vertex and move it to the final position. The procedure is as follows:

```
worldVertex = baseModelVertex * restBoneToWorldMat
deformedVertex = worldVertex * worldToBoneMat
```

There is an easy optimization step, though. Because two matrices can be multiplied together to create a matrix that is a sum of both transformations, I can create one matrix that will do everything:

```
combinedMatrix = worldToBoneMat * restBoneToWorldMat
```

Then for each vertex, I simply multiply it by that one matrix:

```
deformedVertex = worldVertex * combinedMatrix
```

This is the key to skeletal deformation. If you want to have multiple bones influencing a vertex, this final formula can be scaled by the amount of influence, or weight, of each bone. The sum of all the weights for each vertex should equal 1.

What's the Problem?

This process requires each vertex to be transformed by a matrix for every bone that influences it. Clearly, this process should benefit greatly by the use of hardware transformations. However, this really breaks the transformation pipeline. It's obviously possible for a single polygon to have vertices that are influenced by multiple bones. Therefore, you cannot simply set the transformation matrix and draw a polygon as you would normally. An entire character is even more complicated.

In order for this to work, I would need to do the matrix operations first, combine the resulting vertices into a single polygon, and submit that to be drawn. However, the pipeline doesn't allow this. Getting the results of a transformation is not a fast process, as it requires values to be returned from the driver through a mechanism such as feedback.

This is precisely why it's crucial that this type of operation be handled by the driver via the graphics API. It's impossible

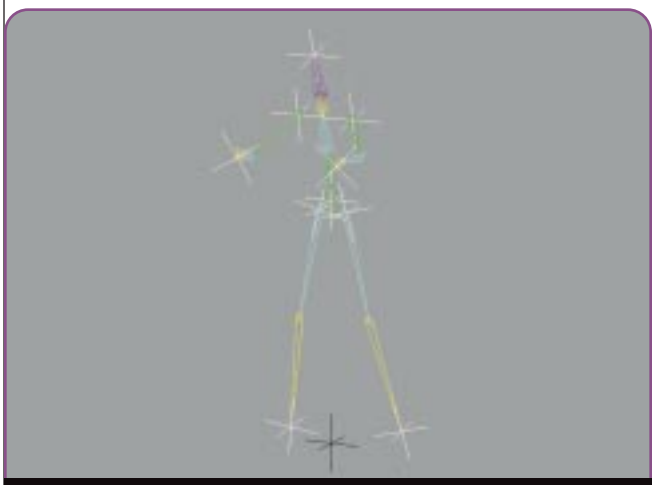


FIGURE 2A. *Our matrix enables us to move the skeleton's vertices to our character's final pose.*

FIGURE 2B. *Our finished model, the beneficiary of our transformation matrix.*

LISTING 2. *Vertex blending, where each vertex is weighted by two bones.*

```
typedef struct tVertex
{
  float x, y, z;
  float    weight;
  D3DCOLOR color;
  D3DCOLOR specular;
  float u;
  float v;
} tVertex;

#define FVF_VERTEX ( D3DFVF_XYZ | D3DFVF_XYZB1 | D3DFVF_DIFFUSE | \
                     D3DFVF_SPECULAR | D3DFVF_TEX1 )

for (int loop = 0; loop < curBone->visuals[0].faceCnt; loop++)
{
    tFace *face;
    face = &curBone->visuals[0].face[loop];
    // There are two Matrices stored for each vertex
    d3ddev.SetTransform( D3DTRANSFORMSTATE_WORLD, face->matrix1);
    d3ddev.SetTransform( D3DTRANSFORMSTATE_WORLD1, face->matrix2);

    vertex = (tVertex *)curBone->visuals[0].vertex;
    HRESULT hr = d3ddev.DrawPrimitive(D3DPT_TRIANGLELIST, FVF_VERTEX, vertex, 3, 0 );
}
```


to perform this kind of deformation without breaking the transformation pipeline. Now many people are against the idea of adding API features for specific effects such as this, but I see no other way to achieve the goal. If the transformation could be handled as a specific DSP operation that was not tied directly to the display, it would be possible to have simple accelerated matrix operations. However, this is not the direction the hardware development is heading.

What's Being Done About It?

Microsoft, while creating DirectX 7, realized that this would be an issue. To solve the problem, they created the notion of "vertex blending." In vertex blending, the final position of a vertex can be determined by the weighted transformation of up to four matrices.

You set up the matrices by creating a `transformstate` with the function:

```
SetTransform( D3DTRANSFORMSTATE_WORLD, matrix1);
SetTransform( D3DTRANSFORMSTATE_WORLD1, matrix2);
SetTransform( D3DTRANSFORMSTATE_WORLD2, matrix3);
SetTransform( D3DTRANSFORMSTATE_WORLD3, matrix4);
```

By using the flexible vertex format, you submit weights in each vertex structure. The number of weights is one less then the number of matrices being used. This is so that the API can enforce the constraint that the sum of the weights must equal 1.

For example, if I wanted each vertex to be weighted by two different bones, I would use code that looks something like Listing 2. There are some problems with this approach, however. First, the API supports blending of between one and four matrices. This leads to a content creation issue. If a particular card supports blending of only two matrices and your content was designed for blending four, you will need to clamp and scale the weights to work. This is yet another restriction for content creators.

Second, you set the matrices for each primitive instead of each vertex. Each primitive submitted to the rasterizer must be composed of vertices that are blended among the same bones. This can be a problem in certain regions, such as the shoulder or waist where it would be quite easy to have each vertex influenced by a different bone.

Finally, to submit primitives efficiently for rendering, the model will need to be sorted by matrix usage. This may mean rearranging your dataset with a sophisticated export utility or custom optimization tool.

While it will be possible to create content to match these restrictions, it won't be easy. Naturally, artists won't enjoy being limited in their methods of weighting, and tools will need to be developed to handle the requirements.

But I Don't Like Direct3D!

Unfortunately, at this time, there's no way to achieve a similar functionality in OpenGL using transformation hardware. The OpenGL community needs to step up and design an extension that will provide access to this hardware capability.

I would prefer an extension that provides more functionality with fewer restrictions. I imagine a vertex accumulation buffer similar to the compiled vertex arrays we have now. In this version, you set a transformation matrix and submit a series of vertices with associated weight values. These are multiplied, scaled, and accumulated. Once all the vertices have been processed, the entire mesh is drawn with this accumulated vertex array.

This system would have no restrictions on the number of matrices in the blend. It would also have the side benefit of enabling many other interesting 3D effects such as morphing. I am not sure if this kind of extension could work with hardware as it exists now but I hope to find out. I'll keep you posted.

What Are the Goodies?

I have provided a couple of demonstrations on the *Game Developer* web site (http://www.gdmag.com). Both of them allow you to manipulate a hierarchical skeleton to deform a 3D mesh. One was created using DirectX 7 and the vertex blending function, the other was created using OpenGL and implements the method I described above. Let me know how you think the algorithm can be improved. ■

Over My Dead, Polygonal Body

Amazing things have been created with computer graphics over the past several years. Visual effects companies are poised to tackle one of the most difficult challenges in computer-generated (CG) imaging: creating a realistic CG human character. However, in my opinion, it hasn't happened yet, no matter how many times I see life-like organic aliens and robotic bipeds trotting all over the screen.

The trouble is, humans are tough to simulate, regardless of how long we take to create the image. It may be obvious, but the difficulty lies in the fact that we are all very familiar with how humans look. We see them all the time. In the morning, I see something resembling a human in the mirror while I shave. Living as I do along the coast, I often see more of the human form than is probably healthy both for my ego and my digestion. In everyday life, we see all varieties of people performing every imaginable action. Each of us is an expert in determining the believability of a CG human form. If the skin looks wrong or the motion looks stiff or the lip-synch is off, each of us screams, "Fake!"

But these technical and artistic problems are solvable. The brilliant artists and technicians charged with making us believe will make sure that it happens. I have thought a bit about the problems this will present, however. I think about John Wayne. What if, while he was alive, he was asked to do a commercial for beer? Perhaps his answer would have been, "Over my dead body!" That used to mean something, that there was no way you would ever get me to do something. A bit of stock video footage and some clever video processing tricks and voilà — John Wayne's dead body selling beer.

Now don't get me wrong. I see nothing wrong with a family member licensing the likeness of a relative for commercial or charitable purposes. I think most people would be glad to continue to provide a living for their family even after they have gone from this life. I just think technology has forced us to reexamine that particular ultimatum. Perhaps it is time for something like, "Over my dead body and virtually extinct telepresence." Kind of loses that Wild West charm, doesn't it?

Bringing Them Back in Real Time

Concerns over these kinds of legal dilemmas are not going to stop me from doing my job, however. As a creator of real-time 3D graphics, I not only face the hurdles which confront my visual effects comrades, but I also need to make these realistic characters move fast. Fortunately, in this task, technology is on my side.

Realistic characters require lots of polygons to make them look realistic. Lots of polygons mean lots of vertices being transformed by an overwhelmed CPU. At Siggraph 1998, I began to see the emergence of transformation acceleration in the consumer graphics hardware space (see "Taking a Break for Siggraph," Graphic Content, October 1998). At that time, I thought it wouldn't be long before we would be able to take advantage of hardware acceleration for transformation and lighting (HW T&L) in our games. Well, Siggraph has come and gone again and we are starting to see this become a reality.

One consumer graphics chip company, Nvidia, has publicly committed to delivery of hardware transformation and lighting with their next generation of graphics chips. There are rumblings that others are likely to deliver on this promise as well. In fact, Microsoft is so certain that hardware T&L will be a reality in the consumer hardware market, they have included support for it in the next version of the DirectX game programming API, DirectX 7. Game developers who are scheduling projects for release in the next production cycle need to consider how their projects will handle hardware T&L.

How Do We Deal with HW T&L?

Fortunately, the driver writers will do most of the work for us. Games using the transformation and lighting pipeline in OpenGL will take immediate advantage of the hardware if it's available. Now, with the introduction of DirectX 7, games supporting this API will also transparently benefit from the new hardware. By using the built-in transformation pipeline in either API, games will get faster.

This will naturally enable games to increase polygon counts without sacrificing performance. It also means that the load on the CPU will decrease,

When not being frightened by the idea of virtual versions of himself, Jeff creates 3D graphics for Darwin 3D. Let him know he is just being paranoid by e-mailing him at jeffl@darwin3d.com.

FIGURE 1A. *Modeling organic objects is a challenge for 3D programmers dealing with hardware T&L.*

FIGURE 1B. *Skeletal deformation systems offer both flexibility and good looks.*

allowing more processor time for game play features, such as artificial intelligence and game physics. And, as to be expected, creating content that performs well on a variety of user system configurations will be more important than ever.

LISTING 1. *Code for the `World->RestBone` matrix.*

```
GLvoid COGLView::GetBaseSkeletonMat(t_Bone *rootBone)
{
/// Local Variables ///////////////////////////////////////////////////////
    int loop;
    t_Bone *curBone;
    tMatrix tempMatrix;
///////////////////////////////////////////////////////////////////////////
    curBone = rootBone->children;
    for (loop = 0; loop < rootBone->childCnt; loop++)
    {
        glPushMatrix();

        glTranslatef(curBone->b_trans.x, curBone->b_trans.y, curBone->b_trans.z);

        // Set observer´s orientation and position
        glRotatef(curBone->b_rot.z, 0.0f, 0.0f, 1.0f);
        glRotatef(curBone->b_rot.y, 0.0f, 1.0f, 0.0f);
        glRotatef(curBone->b_rot.x, 1.0f, 0.0f, 0.0f);

        // Grab the Matrix that is built up to this point
        glGetFloatv(GL_MODELVIEW_MATRIX,tempMatrix.m);
        // Invert this matrix to get the Base->World matrix
        InvertMatrix(tempMatrix.m,curBone->baseToWorldMat.m);

        // Recursive call if the bone has children
        if (curBone->childCnt > 0)
            GetBaseSkeletonMat(curBone);

        glPopMatrix();

        curBone++;
    }
}
```

Since we the graphics programmers can gain this benefit without doing any real work, hardware T&L doesn't seem to affect us much. The programmer will need to conform to a standard pipeline. For the transformation, this doesn't seem to be a big deal. Most 3D programmers are comfortable with matrix manipulation of vertices. The problem is pretty well solved and the only issue of trust revolves around the quality of the driver implementation. Once the hardware is doing the work, the driver issue largely goes away. Most will gladly accept this pathway to hardware nirvana.

Lighting, however, is a much more contentious issue. No one I have talked to is satisfied with the lighting model provided by the OpenGL and DirectX libraries. This issue doesn't bug me that much. I don't know anyone who is ready to give up lighting tricks such as shadow maps and texture-mapped lighting for simplistic Gouraud lighting. While we do need a hardware solution for realistic lighting, this is not it. I advise continued reliance on those lighting tricks. However, as a supplement to pre-computed lighting for dynamic changes or shadow computation, a few hardware point and spot lights couldn't hurt, particularly if they're fast. I can think of a lot of things that a few lights could be used for.

No, simple transformation and lighting is not the problem with hardware T&L. We run into the real trouble when we consider what it means for modeling organic objects.

T&L for Non-Rigid Bodies

Hardware transformation is ideally suited to the display of rigid objects. You first set up the transformation matrix and then submit an object to be drawn. With hardware, it will be costly to obtain the results of the transformation

before the object is drawn. This means everything needs to be set up beforehand.

That's fine for most objects and environments. However, characters do not look their best when composed of rigid objects. As I have explored in a previous article ("Skin Them Bones," Graphic Content, May 1998), creating a character from a single mesh and then deforming it via a skeletal system provides a better solution. By using skeletal deformation, you maintain the good looks of a seamless mesh and the animation flexibility of a hierarchical character system.

Unfortunately, this system requires manipulation of vertex coordinates. Let me review how a skeletal deformation technique works. I have a character in a rest pose in Figure 1a and have created a skeleton for the character in Figure 1b.

In order to deform the mesh with the skeleton, I need to assign each vertex to a bone or set of bones. For example, all of the vertices in the head region should be assigned to the head bone. For now, let me assume that the vertices in the character's head are completely, or 100 percent, assigned to the head bone.

I can determine the position and orientation of the head bone at the rest frame by creating a matrix that represents the transformation needed to move that bone from the origin to its location in the hierarchy. The matrix of each bone is dependent on the matrices of all of its parents, so it is necessary to traverse the entire hierarchy to determine this `World->RestBone` matrix.

Now this matrix will take a vertex and transform it to the location of the bone. However, when I am taking the rest position of the character, I will need to know how to take a vertex in the mesh and transform it back to the origin. Fortunately, since we are using a matrix operation, this is a simple matter of inverting the `World->RestBone` matrix with a standard 4×4 matrix inversion routine. I now have a `RestBone->World` matrix. This only needs to be done once, so I can store this matrix for later use. You can see the code for computing the `RestBone->World` matrix in Listing 1.

I now have the matrix I need to move any vertex from the rest position back to the origin. Now I want to move the vertex to its final pose as shown in Figures 2a and 2b. I can do this virtually the same way. I go through the hierarchy and create a matrix for each bone that will take a vertex from the origin and move it to the bone position in that animation frame. For this `World->Bone` matrix, I don't need to invert it.

I now have everything I need to take a rest vertex and move it to the final position. The procedure is as follows:

```
worldVertex = baseModelVertex * restBoneToWorldMat
deformedVertex = worldVertex * worldToBoneMat
```

There is an easy optimization step, though. Because two matrices can be multiplied together to create a matrix that is a sum of both transformations, I can create one matrix that will do everything:

```
combinedMatrix = worldToBoneMat * restBoneToWorldMat
```

Then for each vertex, I simply multiply it by that one matrix:

```
deformedVertex = worldVertex * combinedMatrix
```

This is the key to skeletal deformation. If you want to have multiple bones influencing a vertex, this final formula can be scaled by the amount of influence, or weight, of each bone. The sum of all the weights for each vertex should equal 1.

What's the Problem?

This process requires each vertex to be transformed by a matrix for every bone that influences it. Clearly, this process should benefit greatly by the use of hardware transformations. However, this really breaks the transformation pipeline. It's obviously possible for a single polygon to have vertices that are influenced by multiple bones. Therefore, you cannot simply set the transformation matrix and draw a polygon as you would normally. An entire character is even more complicated.

In order for this to work, I would need to do the matrix operations first, combine the resulting vertices into a single polygon, and submit that to be drawn. However, the pipeline doesn't allow this. Getting the results of a transformation is not a fast process, as it requires values to be returned from the driver through a mechanism such as feedback.

This is precisely why it's crucial that this type of operation be handled by the driver via the graphics API. It's impossible

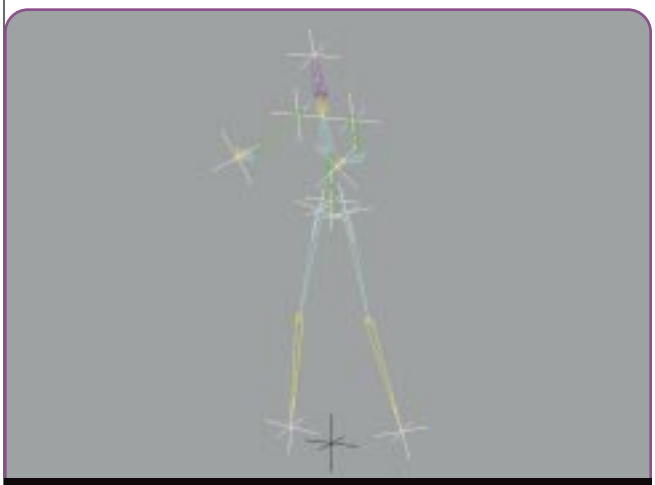


FIGURE 2A. *Our matrix enables us to move the skeleton's vertices to our character's final pose.*

FIGURE 2B. *Our finished model, the beneficiary of our transformation matrix.*

LISTING 2. *Vertex blending, where each vertex is weighted by two bones.*

```
typedef struct tVertex
{
  float x, y, z;
  float    weight;
  D3DCOLOR color;
  D3DCOLOR specular;
  float u;
  float v;
} tVertex;

#define FVF_VERTEX ( D3DFVF_XYZ | D3DFVF_XYZB1 | D3DFVF_DIFFUSE | \
                     D3DFVF_SPECULAR | D3DFVF_TEX1 )

for (int loop = 0; loop < curBone->visuals[0].faceCnt; loop++)
{
    tFace *face;
    face = &curBone->visuals[0].face[loop];
    // There are two Matrices stored for each vertex
    d3ddev.SetTransform( D3DTRANSFORMSTATE_WORLD, face->matrix1);
    d3ddev.SetTransform( D3DTRANSFORMSTATE_WORLD1, face->matrix2);

    vertex = (tVertex *)curBone->visuals[0].vertex;
    HRESULT hr = d3ddev.DrawPrimitive(D3DPT_TRIANGLELIST, FVF_VERTEX, vertex, 3, 0 );
}
```


to perform this kind of deformation without breaking the transformation pipeline. Now many people are against the idea of adding API features for specific effects such as this, but I see no other way to achieve the goal. If the transformation could be handled as a specific DSP operation that was not tied directly to the display, it would be possible to have simple accelerated matrix operations. However, this is not the direction the hardware development is heading.

What's Being Done About It?

Microsoft, while creating DirectX 7, realized that this would be an issue. To solve the problem, they created the notion of "vertex blending." In vertex blending, the final position of a vertex can be determined by the weighted transformation of up to four matrices.

You set up the matrices by creating a `transformstate` with the function:

```
SetTransform( D3DTRANSFORMSTATE_WORLD, matrix1);
SetTransform( D3DTRANSFORMSTATE_WORLD1, matrix2);
SetTransform( D3DTRANSFORMSTATE_WORLD2, matrix3);
SetTransform( D3DTRANSFORMSTATE_WORLD3, matrix4);
```

By using the flexible vertex format, you submit weights in each vertex structure. The number of weights is one less then the number of matrices being used. This is so that the API can enforce the constraint that the sum of the weights must equal 1.

For example, if I wanted each vertex to be weighted by two different bones, I would use code that looks something like Listing 2. There are some problems with this approach, however. First, the API supports blending of between one and four matrices. This leads to a content creation issue. If a particular card supports blending of only two matrices and your content was designed for blending four, you will need to clamp and scale the weights to work. This is yet another restriction for content creators.

Second, you set the matrices for each primitive instead of each vertex. Each primitive submitted to the rasterizer must be composed of vertices that are blended among the same bones. This can be a problem in certain regions, such as the shoulder or waist where it would be quite easy to have each vertex influenced by a different bone.

Finally, to submit primitives efficiently for rendering, the model will need to be sorted by matrix usage. This may mean rearranging your dataset with a sophisticated export utility or custom optimization tool.

While it will be possible to create content to match these restrictions, it won't be easy. Naturally, artists won't enjoy being limited in their methods of weighting, and tools will need to be developed to handle the requirements.

But I Don't Like Direct3D!

Unfortunately, at this time, there's no way to achieve a similar functionality in OpenGL using transformation hardware. The OpenGL community needs to step up and design an extension that will provide access to this hardware capability.

I would prefer an extension that provides more functionality with fewer restrictions. I imagine a vertex accumulation buffer similar to the compiled vertex arrays we have now. In this version, you set a transformation matrix and submit a series of vertices with associated weight values. These are multiplied, scaled, and accumulated. Once all the vertices have been processed, the entire mesh is drawn with this accumulated vertex array.

This system would have no restrictions on the number of matrices in the blend. It would also have the side benefit of enabling many other interesting 3D effects such as morphing. I am not sure if this kind of extension could work with hardware as it exists now but I hope to find out. I'll keep you posted.

What Are the Goodies?

I have provided a couple of demonstrations on the *Game Developer* web site (http://www.gdmag.com). Both of them allow you to manipulate a hierarchical skeleton to deform a 3D mesh. One was created using DirectX 7 and the vertex blending function, the other was created using OpenGL and implements the method I described above. Let me know how you think the algorithm can be improved. ■

Staying in Shape: Low-Polygon Mesh Accommodation

Building and animating high-resolution characters for use in rendered art such as print ads, game cinematics, or movie special effects is fun — the sky's the limit. Your only possible concern in terms of total polygon count is the unwieldy nature of large scenes and the length of render time because of the millions of faces you have to ray-trace. Then again, if you have a pimped-out machine even the mental gymnastics of making sure your model complexity matches your in-view requirements becomes moot. Even so, if your scene contains a distant character that is only 50 pixels high in your 720×486 render, chances are he doesn't have to have 50,000-face boots. High-resolution modeling and animating is a luxury.

When animating characters for real-time polygonal games such as QUAKE 3: ARENA or DRAKAN, "luxury" becomes a four-letter word, and optimization your mantra. More so than in rendered character animation, real-time, low-polygon character animation requires a lot more brain power and planning.

Low-Polygon Defined

I'll begin with defining low-polygon as a character weighing in at under 1,000 faces. Most of the characters I typically work with have between 700 and 900 faces — but without splitting too many hairs, any model with four digits in its face-count just isn't low-polygon anymore. Then again, there are plenty of characters in games today that are over 1,000 faces, but they're usually "boss" or unique characters that are going to dominate a scene and allow for more polygon gluttony on that character's part.

The character in Figure 1, for example, is actually composed of two characters. The big ugly guy is the mount of the actual boss character (sitting on his back) and together they weigh in at around 3,500 faces. Of course, no other enemies can be present at the same time this final boss is on-screen. Otherwise the game would turn into a slide show.

Other factors that determine what "low-polygon" actually means can be the core technology (or game engine) used and level of detail. Level of detail is straightforward enough and consists of adjusting mesh complexity based on its proximity to the viewer. Right now there are several programs, both stand-alone and plug-in, that will optimize your models for you, increase the speed of the game and basically give you more polygons with which to build your characters. As far as what kind of animation technology can drive your characters, there are currently two main types: skeletal animation and vertex deformation.

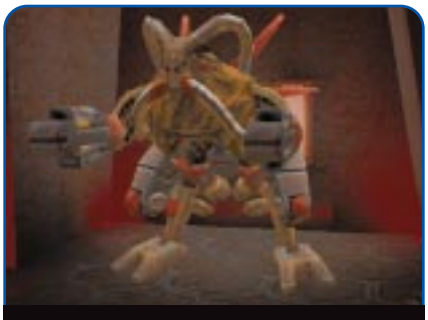


FIGURE 1. *These two characters won't have any on-screen company.*

Skeletal vs. Vertex Deformation

In simplest terms, skeletal animation consists of a mesh being deformed by an underlying skeleton. The mesh itself and its subatomic components of vertex, edge, and face don't contain any animation data. They are attached to the skeleton and are tweaked to certain tolerances as they are influenced by one or more bone. The bones are then animated instead of the mesh itself and the game uses that data to translate motion into the character. Valve incorporated a skeletal animation system into id Software's QUAKE engine to allow them more fluid motion, less media storage per character, and occasional higher-polygon creatures in last year's critically acclaimed HALF-LIFE.

The advantages of a skeletal animation system are numerous to say the least: smoother animation sequences, more realistic and diverse animations, 80-frame walk cycles, and so on. But in the end, the skeletal animation relies on keyframed (or motion-captured) data provided by the artist just like vertex deformation animations. Ironically, most vertex deformation keyframe meshes are derived from animations created using a skeletal system of some sort (just not in the game engine). It's true vertex deformation animation lim-

Paul Steed is a form of computer game artist concentrate. Just add expensive imported beer, hard-to-find microbrewery swill or bottled water to yield a virtually endless supply of ideas, models, animations, and the occasional texture or three. Paul Steed can be found in the upcoming title, QUAKE 3: ARENA. For product information, contact psteed@idsoftware.com.

FIGURE 2. *Players create true real-time animation in QUAKE 3: ARENA.*

its itself by providing a locked pose mesh that essentially becomes a keyframe. It's just as true that skeletal animation limits itself by taking a certain amount of control away from the artist and forces him or her into the constraints of a narrow technological box.

Vertex interpolation, variable frame rate control, and a logically segmented body system via tag representation can make vertex deformation work nearly as well as a full skeletal system and leave a huge amount of control and extensibility in the hands of the artist. Figure 2 shows id's upcoming QUAKE 3: ARENA, in which this technique for animating characters has yielded significant animation enrichment. The biggest benefit of a system like this is that when you move your mouse from side to side, your character's head, then torso, then lower body shift around accordingly. This generates small, ancillary motions created by the player that never had to be keyframed, and in essence become true real-time animation.

Regardless of what animation system the game engine employs utilizing low-polygon models for characters, face-counts are still important for overall game speed. Even with a dynamic level-of-detail system and a higher polygon budget, not paying careful attention to the construction of your mesh is just asking for trouble when you go to animate it.

Look at Those Lines

As you can see in Figure 3, this character has a certain amount of curvature in its design that needs to be conveyed and retained while going through a varied range of motions. Once the model is attached to a skeleton (in this case a biped using 3D Studio Max 2.5 and Character Studio), we can seek out any problems with her mesh integrity as she animates.

Looking at the wireframe of the model, you can see I've tried to keep it as clean and symmetrical as possible. By this I mean making every vertex, edge, and face a useful part of the model. Keeping the model symmetrical in terms of gross polygon usage is key. If it isn't needed to delineate an intentional shape, get rid of it. In essence, the character gets its mass through the shaping of its constituent parts. Of course, the trick is to impart mass in fewer than 800 faces.

So, after the model is attached to her underlying skeleton, I begin animating her. Figure 4 shows how the character looks during her walk cycle.

Not bad. The mesh seems to be

FIGURE 3. *Modeling this character's curves efficiently will be a challenge.*

deforming well enough and the shape is holding up nicely. This particular character carries a weapon of some sort all the time so, I've assigned a wireframe material to it preventing it from obscuring any mesh weirdness.

Now let's look at another animation sequence: her crouched walk cycle, shown in Figure 5. Immediately I see some weirdness at her backside. Given the limited number of polygons, it's hard to get a decent FPB (face per butt) ratio going, so some angularity is to be expected. However, the settings in the animation tool that describe how much influence a particular bone or joint has over the geometry can be tweaked to overcome this. Sometimes, however, the settings are right and the mesh itself needs to be tweaked.

FIGURE 4. *The character's walk cycle — so far, so good.*

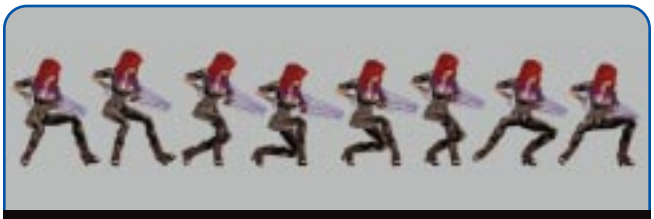


FIGURE 5. *Her crouched walk cycle presents problems.*

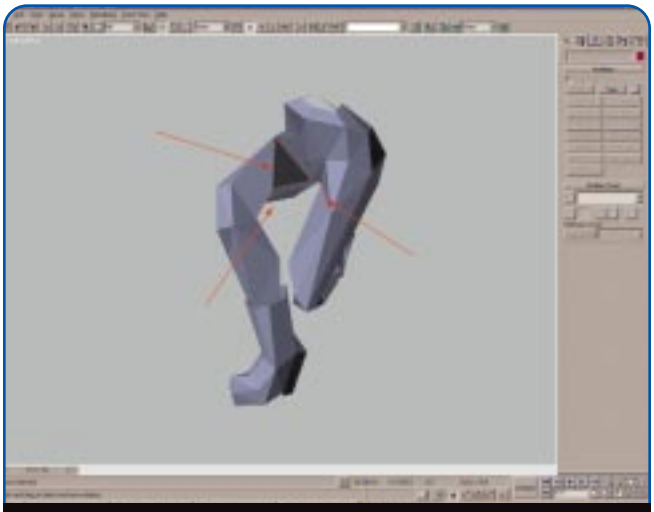


FIGURE 6. *Her inner thighs flatten as she crouches down.*

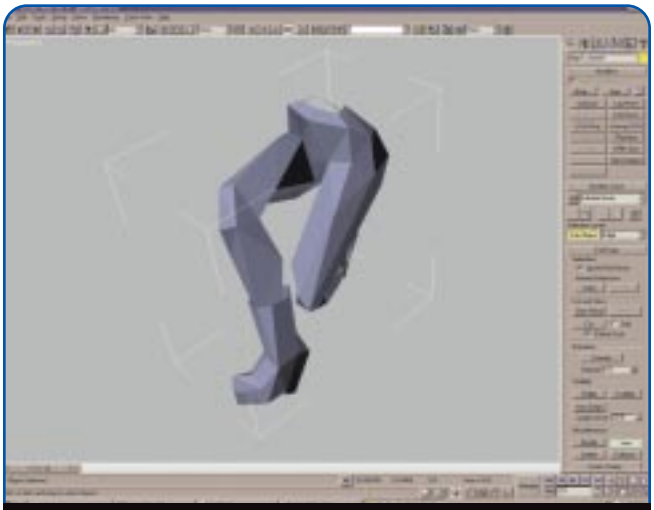


FIGURE 7. *Turning some of the edges alleviates the flattening of the inner thigh.*

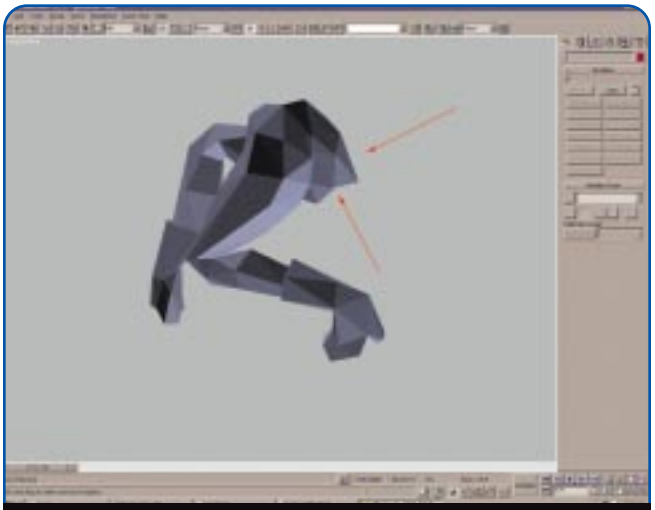


FIGURE 8. *Pointiness in back is an unwanted side effect of fixing the inner thigh problem.*

Look a little closer at the mesh in Figure 6. Notice we've hid everything but the legs and have gone to a flat-shaded, non-textured mode to get a better sense for the deformations of the model. For me, smooth shading hinders my ability to get a feel for how the model is shaping up and I always work in flat-shaded mode when not in wireframe. In the case of this particular model, not only is it losing its rounded shape when crouched, the inner thighs' edges are collapsing. Easy enough to fix — turning some edges will help the flattening of the inner thigh, so let's do just that, as I have done in Figure 7.

Figure 8 shows how it looks better with the edges turned, but what about those pointy cheeks? Houston, from this view, we definitely have a problem. This character is supposed to be a sexy biker/monster-killer/femme fatale who can frag with best of them.

To begin correcting the problem, we need to look more closely at the legs (Figure 9). We take for granted the vertex weighting and bone influences are correct, so the task at hand is to add faces to the hip and leg areas to give her fewer angles and more curves. But first, I'll explain the reasoning behind the current distribution of polygons.

Obviously, frugality is the key here. I've tried to keep the cross-section of the leg at any given point down to no more than a pentagon. The thighs were meant to be strong and healthy, and the knees need extra segments for flexion. To the rear, a few extra faces were put in relative to the rest of the legs purely for shaping reasons and deemed an important enough feature to have a relatively good FPB (not great, but not too skimpy, either). The boots are great since the texture is so detailed and covers up the low number of triangles there. Obviously the problem lies in the area from the mid-posterior to just above the knee.

Figure 10 provides another look at the leg done in cross-section to show what I mean. Notice how the inner thigh lines created by the configuration of faces and edges cause some bad pinching and edge overlap. The two vertices at the top of the thigh can't really move around too much since they anchor the hips, and help keep the mass there. So again, the conclusion stands that we need to put some more faces in the thighs.

Thus, after much tweaking — adding more faces, experimentally turning edges to see what works best, playing around with the vertex weighting, and tweaking some more on top of that — I came up with the solution shown in Figure 11.

Compare the front view of the old and new version of the legs. As I went through the tweaking process, the most interesting change I had to make was the relatively minor adjustments to the height of the vertices that form the hips (1).

FIGURE 9. *Polygons have been carefully allocated throughout the legs.*

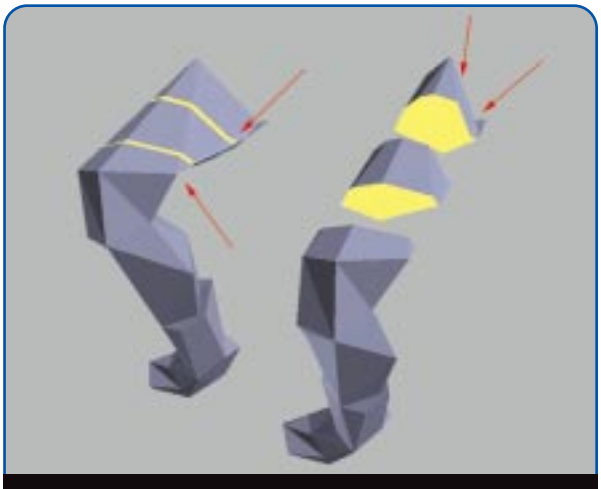


FIGURE 10. *Now we've isoated the problem: some edges are overlapping and pinching at the inner thigh.*

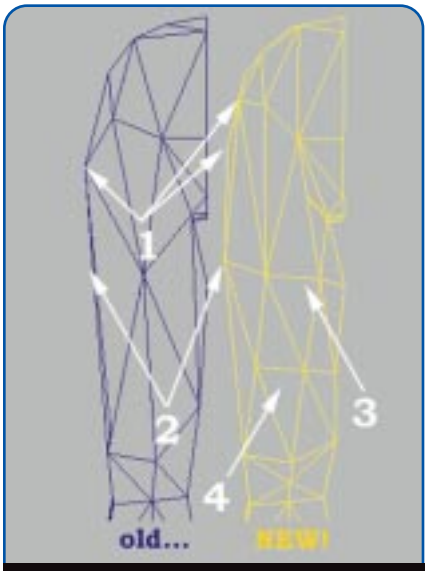


FIGURE 11. *A few small changes will make a big difference.*

This wasn't just a purely form-keeping move to better use the extra polygons I put in her backside, either. This height adjustment was key to making sure that in the crouched position the segments making up the curve of her cheek flexed and distributed themselves evenly enough to look rounded. Again, this may be something particular to the way Physique works in Character Studio, but I remember having the same issues in Alias|Wavefront's Power Animator.

Moving on, also note that when I first built the leg, I simply over-optimized the thigh and compromised its mass by not having a full segment midway up (2). I also had to increase the cross-section circumference to retain mass when the character crouches (3). Next, I had to insert a partial segment just above the knee to support the muscular inner thigh I was trying to impart (4).

Figure 12 shows a full comparison between the old leg (blue) and new leg (yellow). Overall, I had to add more geometry, but more importantly I had to make sure the edges were arranged to avoid inappropriate collapsing. Again, this takes time and experimentation more than anything else, but the fact that there is a limitation on the polygon count makes it more challenging than if I could have added unlimited faces. Referring to our cross-section in Figure 13, we can see the leg is more rounded and full. Now she can crouch walk in style and not get laughed at because her behind is so pointy (Figure 14).

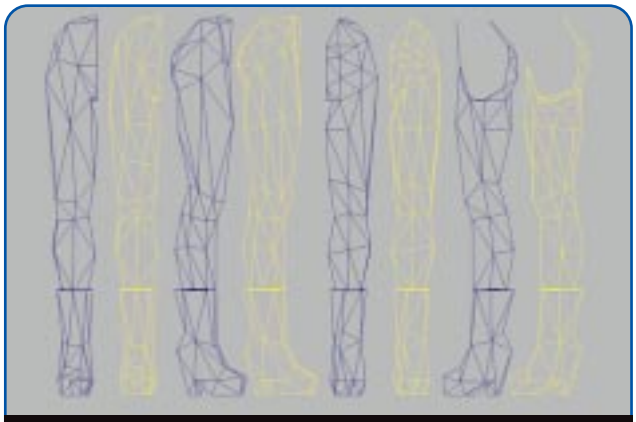


FIGURE 12. *In the end, a little more geometry was added, with careful attention paid to arranging the edges.*

Rising to the Challenge

Creating convincing, good-looking, low-polygon meshes is a challenge. Making sure these models animate well with their polygon budgets requires quite a bit of thought. If I had to come up with an overriding rule of thumb when it comes to ensuring your mesh will accommodate your animations, I'd say it's a matter of convexity. When I tweaked the character's leg to make it retain its mass, my goal was to find edges that were being torqued in such a way that the illusion of mass was being taken away.

I turned the edges to go from a concave to a convex look, but in the case of the upper thigh, I had to insert additional geometry to support its shape. Viewing your model in flat-shaded mode makes finding concave edges relatively easy, but very crucial. When reviewing and testing your models for proper deformation, definitely seek out those edges and turn them. Unwanted dents and divots in your mesh break the illusion of mass very quickly and just plain make your model look bad.

I know attaching a mesh to a skeleton and spending hours getting your vertex association just right is a difficult thing to throw away. I cringe every time I realize I have to detach the mesh and tweak it. Not only is reattaching it a pain, but reassigning UVs and getting the texture (if it's already been done) to line up makes for hair loss and lack of sleep. However, if you have to perform massive reconstructive surgery on a part of a model that isn't quite working out to make it better than it currently is...tough. Unless you're going gold the next day, never settle for what the computer will give you — only settle for exactly what you want. It's what separates the men from the boys and the women from the girls. ■

Where's Mel?

Now that DRAKAN has shipped, Mel Guymon is taking a few months off to recover. He will return in January 2000.

FIGURE 13. *After the changes, the thigh is rounder and fuller.*

FIGURE 14. *At last, our character can crouch with confidence.*

Interplay OEM: Little Bundles of Joy

Recently, Mattel Media announced that it was teaming up with Patriot Computers to launch Barbie- and Hot Wheels-branded PCs for children. These lunchbox PCs (my term, not the manufacturers') would cost around $599, and include a library of education and entertainment software from Mattel Media and its recent acquisition, The Learning Company. No prizes for guessing which computer would have the girl software, and which the boy software.

Obviously, bundling software with hardware is a symbiotic process that should benefit both the software and hardware partner, but it's also a specialist sales channel that needs to be approached with care.

Everyone a Microsoft

The greatest bundling machine in the computer business is Microsoft. Not only does the company have its operating system bundled with every PC sold, but you can almost guarantee that there are other Microsoft goodies on your hard drive, and even the odd interactive media product. In very simple terms, Microsoft makes the software that sells the hardware, and in return, the hardware vendors have had little desire to go anywhere else for software. Fortunately, Microsoft has also driven PC technologies by supporting features in its operating systems that have made it possible for hardware innovations to flourish.

Game developers may not be able to offer the same breadth of opportunities as Microsoft does with Windows, but they do hold a key, which can unlock higher-end technologies such as 3D graphics and surround sound audio, and they also help Intel and AMD sell ever-greater performing CPUs. Leading the charge in entertainment software OEM deals is Interplay OEM. In fact, Interplay is the only one of the major publishers that has set its OEM business as a separate subsidiary. Interplay OEM should stand alone; the company counts among its customers LucasArts, Take Two Interactive, Fox Interactive, Westwood Studios, Virgin Interactive Entertainment, and Gathering of Developers. It's the model of how OEM sales should be built into a game publishing business.

Additionally, Interplay OEM also handles licensing and merchandising activities on behalf of Interplay and Shiny Entertainment, including novelizations, strategy guides, and other merchandise tied to Interplay's game properties. There are also arrangements for royalty-based revenues from licensing arrangements and the sale of products by third-party distributors in international markets.

Bundling for Profit

Jill S. Goldworn has served as president of Interplay since December 1996, and has handled OEM contracts for the last ten years. Unlike some game developers and publishers that view OEM sales as part promotional tool and part incremental revenue stream, Goldworn has a strong belief that the OEM sales channel is a major revenue contributor.

Goldworn says, "OEM sales are for pure revenue purposes and, also important, getting access to cutting edge technology early. It's the same as in the console market — if a developer gets a hold of the next-generation technology early enough, it is positioned to take advantage of the console at launch. We'll actually put in support for some technologies before the traditional retail channel establishes a need for it — for example, DVD. We've been developing DVD titles for a number of months. So, we've actually developed DVD titles for OEM, which retail will take to the market later."

The benefit of marrying the revenue potential of OEM sales to the technological advantage it can give a developer isn't lost on companies such as France's Ubi Soft, and Rage Software, based in the U.K. Ubi Soft's TONIC TROUBLE, and previously, POD, have achieved unit shipments in excess of one million by piggybacking behind the Pentium II and Pentium III launches. POD was probably the standard-bearer for MMX-enhanced games, and while it may have done well in OEM channels, there never was any retail MMX market to speak of. In 3D, Rage's

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.

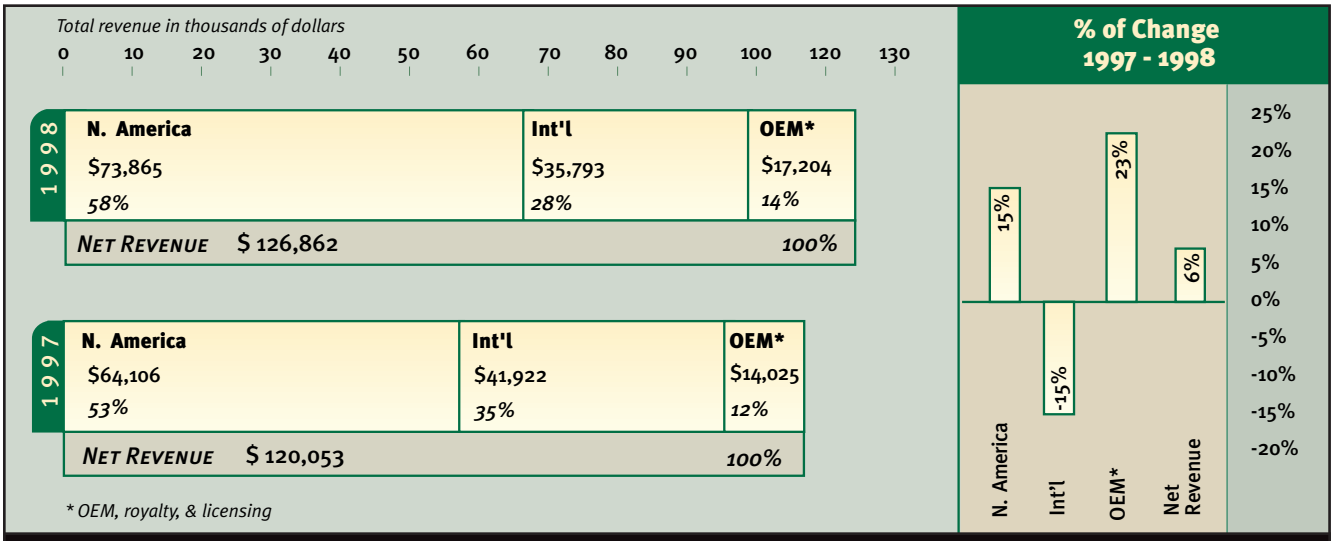


CHART 1. *Interplay OEM's performance relative to overall revenue achievements, in thousands of dollars and percentage points. Interplay OEM also provides licensing and merchandising deals, but it can be safely assumed that the bulk of its sales are from straightforward bundling deals (source: Interplay).*

INCOMING has been among the best selling OEM titles in recent years, probably constituting a third of Rage's total revenues. Now, the company's follow up, EXPENDABLE, is following in its footsteps, and looks as if it too will come out equally strong in Pentium III lineups in 1999.

Goldworn herself has evangelized the OEM channel to the game industry drawing other publishers into the Interplay OEM fold. "Our whole model is to educate publishers about technologies that will eventually become a revenue stream for them," she says.

LucasArts used to think of OEM sales as a small contributor to its revenues, and not worth focusing on. However, after five years of working with Interplay's OEM team, the company is now putting an OEM SKU on its release schedule. Whereas in the past LucasArts would take months to build an OEM version of a title, it now has an OEM support and development structure in place almost simultaneously releasing retail and OEM SKUs. For companies such as LucasArts, it makes good business sense to be in OEM, but it is also valuable strategically. The developer gets early access to technologies, and the title gets automatic exposure in the hardware vendors' marketing and sales promotions.

Of course, the hardware technologists need game developers more than they'd care to mention. Intel, AMD, 3dfx, Nvidia, Diamond, Creative, ATI, and Matrox have all courted developers with technology, free hardware, and sometimes, cold, hard cash, in order to get a game or games ready for a new product launch. It also helps when a company such as Intel recommends the cutting edge titles it favors to its PC OEMs. It's a win-win situation for all parties, but developers have to be aware of the problems of jumping on the cutting edge of technology, too.

Working with new technologies often means that a game title has to be modified for it, in addition to being maintained for the existing installed base. Everything from code to art work has to be refreshed in some cases, and the results could be slippage in shipment times, or even lost development time if things don't go according to plan. In other words, the risks of working with any new technology don't go away when an OEM is involved, regardless of the financial carrot being dangled.

Delays in shipment of a title, or the expectation that an OEM release will translate into a big retail launch, are often the two biggest problems developers face. If a developer is tied into an OEM release in order to get a launch for a new title, any delay can have serious repercussions. Even a successful OEM product doesn't guarantee a similar impact in the retail channel. Ubi Soft's POD proves this. Therefore, developers should plan for OEM in much the same way as they plan for any title.

An OEM Strategy

With the development hiccups of OEM opportunities having been overcome, there are still numerous ways in which a company can get a title bundled with hardware:

STRAIGHT: The straight bundling deal requires the developer or publisher to deliver a CD for inclusion by the hardware vendor with its products. The deal is often subject to a certain minimum quantity, or staggered so that different discounts apply as the vendor ships increasing numbers of the bundled software. The straight bundle deal is probably the easiest to track because finished goods are being shipped, and revenue generated directly thereafter. However, while straight bundling deals are great if you are in the position of an Interplay OEM, and have a strong catalog of titles to offer, a smaller developer will often find them more difficult to come by. Still, there is always room in the OEM market for a title that defines a new technology and gets consumer recognition of it.

TARGETED: A developer or publisher may choose to target a particular 3D graphics chipset, or CPU product. Ubi Soft is doing it with Matrox, and Interplay

OEM has done it with 3dfx in the past. In effect, the modified OEM title will only work on the hardware vendor's particular product. For example, the title might work with 3dfx's Voodoo 3, but not Voodoo 2. It's almost an ideal bundle arrangement because the hardware vendor gets to launch big on a product and use a specific title for its platform, while a developer gets the benefit of the exposure the hardware brings, but can still look forward to exploiting opportunities in other areas of the market and on other hardware platforms. Intel, AMD, and any number of 3D graphics and audio guys have a vested interest in getting a unique experience for their products.

ENCRYPTED: Some vendors have attempted — with little success thus far — to ship encrypted, time-locked versions of their titles with a hardware product, hoping to get the benefit of the full sell when a user buys the product-unlock key. This is more of a marketing strategy, and less of a bundling arrangement, but it is something that has been tried in the OEM market before. The results for game companies have been poor, with response rates being equivalent to a direct mail campaign, in the region of one or two percent. So it works for AOL disks, but not for games. Still, many hardware vendors are looking at ways of creating bundling arrangements that will allow them to get a slice of the retail sale of a title. The most obvious possibilities are going to be on the Internet. As more hardware companies develop their direct sales on the web, they are looking for ways to leverage their products off of software and content. In many cases, selling a package of goods, such as a graphics board and a handful of titles, is better than trying to sell the individual components of the deal separately.

LIMITED: With some popular titles, limited versions, or versions containing specific levels but not the full retail complement, are used in OEM bundles.

LICENSED: Many vendors are moving to a model whereby they license content, therefore not requiring any finished goods from the developer. As is the case with most PC OEMs, bundled software is preloaded on the hard drive, and the cost of reproducing a number of bundled software CDs is saved.

As for the revenues, bundled titles can fetch anywhere from as little as 35¢ to as much as $14 per title. The minimum quantities required to qualify for a bundle deal varies. I have seen deals done for 12,000 unit commitments on some high-end peripherals, to multi-million-unit commitments on some CPU products. For most of the major publishers, once a product's revenues fall under $1 per unit, the OEM sales lose almost all their appeal. Publishers will change pricing depending on the age of the product, or the hardware vendor they are dealing with. Whatever the price or the unit commitment, new technologies such as faster CPUs and faster 3D graphics need games to turn consumers on to them. It's also worth knowing that these new hardware technologies are more profitable for the vendors, thus leaving room for an aggressive OEM sales company like Interplay OEM. ■

Breaking the Sound Barrier

How to work with a third-party sound designer

BY AARON MARKS

Sound effects are an integral part of games, equal in importance to artwork, music, and game play. And I'm not just saying that because I create them either — I'm also a game player, so I understand their impact from that perspective, too. Sounds are designed to absorb the player into the game world, to make it believable, entertaining, and satisfying. The process of determining the appropriate sounds for a given game situation, creating them, and implementing them in the game isn't always a painless experience. Both sound designers and game producers need to understand each other's professional needs and responsibilities so that the sound design process becomes less grueling to both parties. This article describes the process of determining what you will need from a third-party sound designer, what that person will need from you, and will briefly describe the process audio contractors go through to create high-quality sound effects.

Aaron Marks (aBmajor@aol.com) is a sound designer, music composer, and owner of On Your Mark Music Productions. Look for more of his life story at: http://members.aol.com/aBmajor.

illustration by Stuart Bradford


SWOOSH
TWEET
TWEET
BEEEP
SCREETCH
VROOOOM
KABOOM
TICK
TICK
TICK
BZZZZZZZ

Where Does It All Start?

Joey Kuras, sound designer for Tommy Tallarico Studios, has personal credits on more than 60 games, and he recently worked on the James Bond game, TOMORROW NEVER DIES. Early in the production cycle, he was given a list of effects needed for the title. He designed and delivered more than 200 sounds per the developer's request, only to have approximately 90 percent of them discarded as the production matured. He ended up recreating them later in the project. On another project, he received an unspecific and vague list of sounds. The request for a "splash" sound had little meaning to him. Was it a rock making the splash? A person? A 400-pound object or a four-pound object? Was the body of water a bathtub, a pond, or an ocean? No one he talked to was sure and they ended up waiting far into the project to solve the mystery.

This is a lesson for all of us. Early concepts of art and game play have a tendency to change and continuously evolve, so sound design at the outset of a project is usually a grand waste of time. Producers have the difficult task of trying to determine the audio needs of the game early in the development cycle, and if a contracted sound designer is used, the producer must find one whose skills and credits match those of the game being developed and negotiate the contract with that individual. When you are spending up to $30,000 for sound effects, you want to get your money's worth.

The producer may have to take a long-range look at what sorts of sounds the game may need (general Foley sounds, imaginative or "far out" sounds, and so on). If the game is to have a wide range of settings and characters, it can be difficult to imagine what this large bank of effects will consist of. Therefore, it's important to bring the development team together to begin thinking conceptually about the game audio as early as possible. If you've decided on a sound professional, bring him or her in on the discussion and listen to what the sound designer's experience has to say. A lot of time can be saved this way and the process will have more grease for a smooth ride. Don't be tempted to jump into the audio implementation details, however — starting the creation of game sound effects too early in the process often leads to major headaches down the line, as I'll explain later.

Usually, when a game is far enough along (at the point when characters, movements, and a defined game play model are present), the sound designer should enter the picture. By permitting sound designers to meet with the development team, view some rough game levels, and perhaps see some animation ideas, their idea machine can begin to churn out possible routes to take. Also, asking a few

specific questions will bring a direction and necessary information together to get off to a smart start.

Contracting Out the Work

The actual task of finding a third-party contractor can be as arduous as creating the game itself. Unless you've worked with a particular sound designer in the past and you're comfortable using him or her again, you'll need to take care of some advance work. Even before the project is put out for bid, a media buyer (often the producer's role) can do homework. Investigate various sound design companies beforehand to stay ahead of the game. Web search engines can help, developer resource web sites are prevalent, and the numerous unsolicited e-mails, inquiries, and résumés can (finally) be taken advantage of. Request a sound designer's current demo reel, references, and examples of past work, and keep this data on file.

When the time does come to start looking at bids, the producer alone, or with several of the team members, sits down to evaluate submissions. Generally, they are looking for outstanding work, creativity, a shared vision, reliability, experience, and someone with whom they feel they can work for the length of the project. After the field has been narrowed to a couple of choices, pick up the phone or invite them over. It's a good idea to talk to the candidates either by phone or in person before any final decisions are made. Check their production schedules to ensure they will be available (some busier sound people are booked two to three months in advance), see which one you feel is best at communicating and receiving ideas, and get a sense of whom you can get along with.

Moving on from the courtship stage toward project commitment necessitates that both parties bring their interests (and sometimes lawyers) to the table and work out an agreement both sides feel comfortable with. Typically, though, negotiations for sound design work are fairly simple. More complicated negotiations come up when music creation is also part of the deal, because that can also involve hashing out ancillary rights, payment for different SKUs, bonuses, property rights for soundtrack releases, and so on. Spend a little time working out an equitable agreement and get the business out of the way so you can focus on the creative aspects of great development.

Prices for sound design services vary per contractor, as each contractor has different overhead costs to meet. Those with more experience can, of course, demand more, and their experience is usually worth the price. Rush jobs, special requests, and other tasks assigned to the sound designer (such as auditioning, hiring, and producing voice talent and their sessions, abnormal amounts of revisions or change orders, and so on) can increase the price, too, so try to plan ahead. The bottom line is that costs are definitely negotiable, but don't expect the contractor to work for free or below their expenses. For more information about payments and contracts for sound design, see the sidebar on p. 44, "The Audio Development Agreement."

Questions Sound Designers Ask

Around the time a producer is trying to wrap up contract negotiations with a sound designer is when the sound designer is going to want to know the nitty-gritty details about the project. This is the point at which "clear and concise" can mean the difference between complete audio bliss and a sound disaster. A good sound designer will want specific information about the game. As the producer or team leader, be prepared to answer the following questions clearly and concisely:

FOR WHAT PLATFORM IS THE GAME INTENDED? This will suggest what type of playback system the consumer will use and the confines the final sound effects will rest within. As a sound guy, I mix to several playback systems — from "el cheapo" multimedia speakers you buy at the grocery store to high-end studio monitors. My interest is to make my sounds work well with them all, but my main focus is on the system the majority of people will be using.

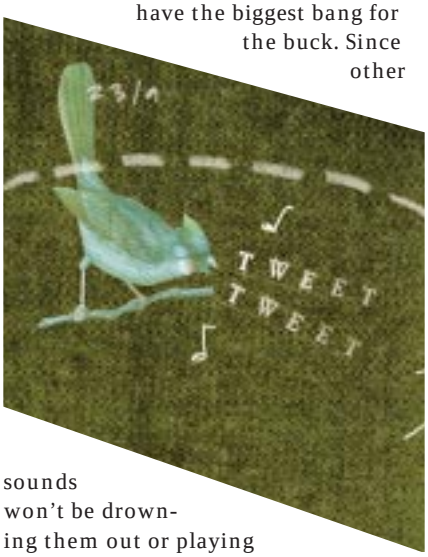
FOR WHAT GENRE IS THE GAME INTENDED? You want the music and sound effects to follow the spirit of the game, so make sure you communicate this to your sound designer, including the feel of the game, what genre it falls into, and what similar games are currently on the market.

WHAT SAMPLE RATE, BIT SIZE, AND FILE FORMAT ARE PREFERRED? Should the audio be in stereo or mono? The development team should have done all of its homework to determine how much space the graphics and sound will be allotted. This information will help decide the sound quality level and within what parameters the sounds should be created.

WILL SOUND EFFECTS BE ALTERED BY ANY SOFTWARE OR HARDWARE PROCESSORS? Additional processing by a game engine will determine to what extent certain sounds are processed beforehand by the sound designer (if a sound designer applies reverb to a sound that the developer had planned to apply reverb to in the game, that could be a problem). For example, driving games often apply a reverb effect to sounds. This is the kind of information the sound designer needs to know at the outset. Decide as soon as possible if there are plans for this type of processing and communicate them, so that the sound designer doesn't overprocess any files.

ARE ANY AMBIENT SOUNDS NEEDED? You probably don't want to distract the player with silence. If the game will use background music, tell the sound designer — designers won't know this fact unless they are also the composer. This question may jog a producer's memory and alert the team to the fact that more than just event-driven noises are needed.

WILL CERTAIN EFFECTS HAVE PRIORITY DURING PLAYBACK? There can be instances during a game (a player unlocks a hidden door, stumbles into a trap, or is attacked by a villain) when a single sound punctuates the moment. All other sounds become irrelevant and this one sound takes priority. These are the ones you want to have the biggest bang for the buck. Since other sounds won't be drowning them out or playing over them, you won't have to consider whether other effects can be heard at the same time. It's critical to get this type of sound perfectly, and by alerting sound designers to these effects, they'll know which ones to pull out the stops for.



WILL THERE BE ANY VOICE-OVERS OR SPEECH COMMANDS THAT NEED TO BE HEARD? Similar to the way vocals must stand out in a song mix, any vocals in a game must be heard easily by players. A sound designer can be involved with processing speech via an equalizer or the volume controls to ensure they can be heard and understood over the other effects.

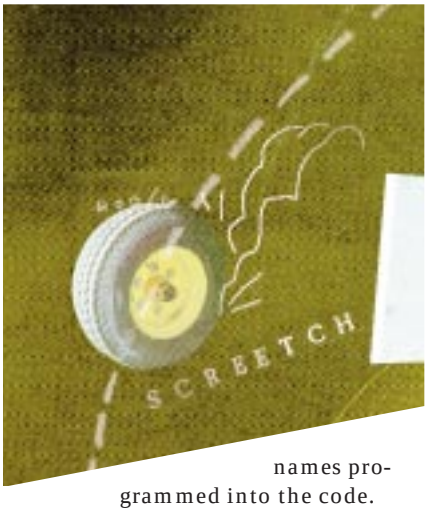
ARE ANY NARRATIVES NEEDED? Will there be background sounds to accompany narration? Narratives fit into the sound recording category, and generally anyone capable of sound design can also record narration. If you already have narratives recorded, the sound designer can usually transfer these recordings into digital files, maximize the sound, cut them to length, and add any additional background or Foley sounds. If narratives are to be recorded, sound designers need to know if they have to provide the voice talent so they can budget accordingly. A good question to ask prospective sound designers is whether they have any experience directing narrative sessions, and if not, make it clear that the producer will fill that role.

ARE THERE ANY SPECIAL SOUND CONSIDERATIONS? Is the game intended to be an audio trend setter, and use technologies such as Dolby Surround Sound or DTS? Are you planning to advertise the game as having "cinema quality" sound? Knowing this ahead of time could be an important safety tip for the sound designer's longevity in the business.

WHAT TYPE OF MUSIC, IF ANY, WILL PLAY AS THE SOUNDS ARE TRIGGERED? This would give the sound designer an indication of what other sonic activity will be happening during the game. If the music will be a soft orchestral score, you might want the sound effects geared to that mood, and not sound too obtrusive. If a rock soundtrack will be played, then harsher sounds and careful manipulation of an effect's higher and lower frequencies will ensure these stand out. The sounds should all work together to enhance game play, not aggressively compete with one another.

ARE ANY SOUND RESOURCES AVAILABLE TO THE SOUND DESIGNER FOR LICENSED MATERIALS? ALIEN VS. PREDATOR, STAR TREK, and SOUTH PARK, for example, are games based on film or television properties that were produced under licensing agreements. If the publisher or developer has secured use of the actual sounds from these works, sound designers need to know if they have it at their disposal to manipulate for the game, or if they are expected to recreate it themselves. While your sound designers may not have an actual hand in creating them originally, they are equipped to convert them to the proper formats and sample rates and need to know, for planning purposes, if this service is desired, too.

ARE ANY SPECIAL FILE NAMING CONVENTIONS REQUIRED FOR FINAL DELIVERY OF SOUNDS? If the development team is overly organized, or if they waited until late in production to bring a sound designer on board, they may already have file



names programmed into the code. While renaming files is not a big deal, it may help cut down on any confusion when delivery is made if they are already appropriately named. The developer should make this need clear or define an acceptable method.

Getting to Work

Once the sound design contractor has been hired, the previous questions have been answered, and the appropriate nondisclosure agreements have been signed, one or two people on the development team should be assigned as liaisons to the sound designer, who are responsible for communicating the project specs and signing off on work. This ensures clear and effective communication, which is the key to obtaining sounds that match the team's vision.

If you're using a local contractor, have them stop by, meet the rest of the creation team, and discuss the game. If the sound designer is unavailable, send copies of artwork, storyboards, and any story text already written. If there are any rough animations, movie promos, or even an early version of the game available, send those, too.

There are several ways to convey sound effect needs to the sound designer. You could create a sound list that describes each sound, what it will be used for in the game, and the requested sound duration. This is really just a wish list, because often the entire list isn't completed for the game — changes in the game see to that.

A producer can also indicate where sounds are needed by giving an alpha version of the game to the contractor that uses place-holder sound effects (general effects-library sounds or effects taken from other games). Placeholders can, of course, simply be spoken words created by someone on the team — for a game I'm currently working on, the producer inserted audio files of himself saying words such as "click," "bonk," "explode," or "shot" that are triggered by the appropriate game event. As I play the game, every time I hear his voice, I create a sound to match the action.

With the preliminary action accomplished, the sound designer has a solid idea of what the game is looking for and sets out to get things organized on his or her end. Jamey Scott, sound designer and composer for Presto Studios (developers of THE JOURNEYMAN PROJECT series, GUNDAM 0078, and others) believes putting together the initial palette — finding sounds that will mix well together — is the most important step. He uses the E-Mu Emulator 4 sampler in his sound design process, and for each game he develops an entirely new sound palette to keep them original. Scott feels that using a sampler has advantages over straight computer files and sound editors. "Layering sounds internally in the E4 works very well for me," he says, "more so than doing it on a computer. That way, I can save my banks as a palette rather than having sources in various folders all over the computer. They are all looped, equalized, and noise filtered all to my specifications. Plus, returning to them to make any changes is a simpler task."

Soapbox Time

Producers and developers who know absolutely nothing about sound design tend to place undue demands on the sound designer, or sim-

A Sound Is Born

Here's an example of how one simple sound effect is created. The project I'm currently working on is a space strategy game, in which units are maneuvered in formation to battle against other players. It is a PC game with final sounds to be delivered as 22KHz, 16-bit .WAV files. I'm creating a sound which is triggered when a shielded unit is fired upon.

The shield sound should have an "electric" quality to it — a controlled surge of energy that might sound as if it were deflecting a shot from a laser weapon. I wanted it to sound unique, so I stayed away from stock library effects and used one of my synthesizers for initial inspiration.

I ultimately settled on a patch, similar to the keyboard sound in the Van Halen song "Jump," and recorded four seconds of a three-note chord. I saved it into my audio editing program, Sonic Foundry's Sound Forge, as a 44.1KHz, 16-bit stereo file. Experimenting with a few different effects processors, I found a nice Doppler effect in another program, GoldWave. I edited an existing patch to give it a quick one-second Doppler increase with three seconds of Doppler decrease. Back in Sound Forge, I pulled up a radio static sound file (which gives it that "electric charge" feel), ran it through a 1Hz stereo flange effect, and equalized it to increase the high frequency range. I then cut that file to four seconds to match the manipulated keyboard sound and mixed the two sounds, keeping the static barely perceptible. I gave the new mixed file a one-second fade-in, and faded out the last two seconds with a linear fade. Now it was beginning to sound like something. I normalized the file to maximize the sound, adjusted for any abnormal level peaks and finally saved the new file as SHIELD.WAV.

Later, the producer wanted a dull, metallic clank mixed in to give the player some distinction between a shield hit and a hit to the unit's space suit. I pulled up a nice clank sound, used the equalizer to get rid of most of the high frequencies, and mixed to the shield file. All was well; the producer was happy.

Converting down to 22KHz is simple. Utilizing the resample feature in my audio editor does the trick nicely. Because some of the higher frequencies get lost in the conversion, I usually adjust the equalization to compensate. After another check on the levels, the effect is ready for the game. Total time spent creating this one sound effect: two hours, 15 minutes.

ply ask for the impossible. It is tremendously frustrating to negotiate and work with those who don't have at least a basic concept of what goes into our process. One original sound can take more than two hours to create — we're not just taking clips from an effects library disc and converting it to the needed format. (To get an idea of the sound creation process, see the sidebar "A Sound is Born.") An entire game can be done in two weeks — with much pressure on the sound designer — but it can take a month or two just as easily.

Ensuring the Best Audio Results

When a sound designer devotes his or her time exclusively to your game for the duration of the project, it helps ensure consistent game audio. Ask prospective sound designers up front whether they will commit solely to your project. The second concern to address with your sound designer is that of quality assurance. All of the sound designers and many of the producers I've talked to insist that the sound designer listen to the sounds in the actual game. Typically, this happens around the time the game goes into beta. It's not uncommon for effects to sound great in the studio but not so hot (too loud, soft, long, or short) once they're synchronized with the action in the game. While analyzing the audio in the beta version of a game he was working on, Joey Kuras discovered a programmer on the project had taken one of the effects — the sound of footsteps — and bumped up the volume. What were intended to be subtle, barely discernable Foley effects turned into a loud series of crunches. Thankfully, Joey's screening session caught the problem and corrected it in time.

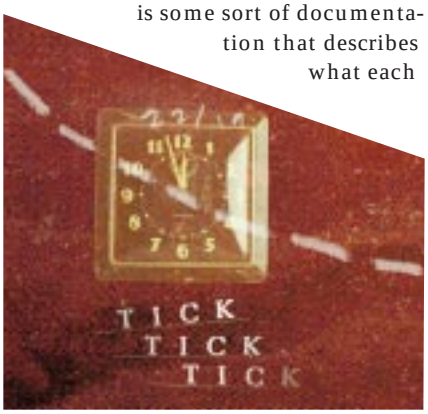
Specify to your sound designer that you want your sound effects created in the highest quality possible (which today is usually 44.1KHz, 16-bit stereo). Because new technology is now becoming more mainstream, some development teams may soon opt to go even higher to 96KHz, 24-bit audio. Why? Because you want to develop the sounds in the highest fidelity then convert down to what is needed for the game. If a game needs 22KHz, 16-bit stereo sounds and it is later discovered it can fit in 44.1KHz, 16-bit stereo sounds, it may already be too late for the sound designer. Attempting to convert up almost always adds unacceptable noise to a recording — you just can't do it. You usually have to start the whole recording process again from scratch.

Not too long ago, one game company decided to create a television commercial for its game and intended to use the original effects from the game. They contacted the sound designer who worked on the project and requested their sounds in a CD-quality format. Unfortunately, he didn't have them at that sample rate and proceeded to spend a few sleepless nights recreating them. It would have been just a few minutes of work had they already been available.

Presenting the Final Work

Delivering the final sound effects to a client is not usually just a matter of sending a CD or e-mailing the sounds and saying, "Here they are!" When presenting finished effects to producers, some sound designers (myself included) send more sounds than were actually requested by the client. I work up several effects, let the producer in on the process, and give clients the chance to choose the effect that matches their vision. Some have subtle differences, changes in length, layering, or effects processor settings. Other sounds are completely different but still evoke a similar emotion or idea. This procedure actually serves dual purposes: it gives the producer radical ideas that just might work, or it makes other effects stand out and sound that much better.

Occasionally included, if it isn't already obvious by the file names, is some sort of documentation that describes what each sound effect is for. This key will save some headaches for everyone involved and score some points for the organized sound designer.



Avoiding Production Nightmares

There are plenty of ways that things can get screwed up when working with a third-party sound designer. For example, the producer may not choose the proper adjectives to describe the game and, in turn, the description may not mean the same thing to the sound designer. Or, the production could go through dozens of design changes, causing the sound designer to lose interest and quit. The list of potential problems is endless. We've all had our own experiences which we'd rather forget, but it's important to learn from them, if for no other reason than to prevent history from repeating itself. Here are some true horror stories from the trenches.

Mark Temple, owner and executive producer of Enemy Technology, has many games to his individual credit and he's currently at work on his company's first game. One of his biggest pet peeves is trying to work with sound designers who are not computer literate. Yes, believe it or not, there are still people out there who don't know much beyond their immediate sound applications. He has made several treks during hectic schedules to visit a sound designer just to get a copy of a game running. He's also had to instruct them how to zip and unzip files, attach sound files to e-mail, or use a modem to connect to the company BBS. So now when hiring people, he makes it a point to ensure first that they know their way around a computer.

Another time, Mark's sound designer left a project in the middle of the contract, with half of the milestones completed and half of the sound effects budget. A new sound artist was quickly brought in, but the effects had a different quality and it became difficult to match his sounds to the previous work. Reluctantly, they opted to start from scratch, which forced them to spend more than they had budgeted and also broke their schedule. People leave in the middle of projects all the time for various reasons, and there are as many different ways to prevent this as there

are reasons to leave. But honest communication and understanding of the creative processes helps. Contract points, which give the proper incentive (such as increasing milestone payments over the course of a project, with the largest payment for the completion of all work) are another option, but overly aggressive contracts that withhold too much money until the end can spoil things, too. Try to find the right balance.

A potential contract sound designer, whose work is outstanding, wanted to work on a particular game, but didn't have the right equipment for the particular project, nor the money to buy it. And because the development team didn't have the resources to help him with this purchase, he didn't get the job. If you find yourself in this position, talk about the situation with the prospective sound designer and see if you can come up with a creative solution before you pass up the deal. Developers have been known to loan or buy equipment for the contractor, depending heavily on how badly the contractor is needed and the cash flow of the developer.

The idea of "one-stop shopping" for a sound designer who can also compose and produce music is appealing. It's an opportunity to hire one fewer person, saving time and money. However, when Mark began his search for just such a person, he found few available. What may have helped this situation would have been to have the producer looking in the right places (no offense, Mark). If they had done any previous research, they could have had a list of names and demos already at their disposal. But since that didn't appear to be the case, there are plenty of online sites that cater to this very thing, some of which are listed at the end of this article.

Here's my horror story. A while back, I replaced a sound designer late in the production cycle of a fantasy game, after the producer became dissatisfied with the previous contractor. It seems the former sound designer was missing his milestones by ever-increasing amounts, and the overall creative quality of the work had declined rapidly. As an example, for some of the spoken magical spells used by the various wizards in the game, he recorded profanities and simply played them backwards. While his "shortcut" wasn't recognizable to the average listener, someone with audio editing software could reverse it and a lawsuit could develop, something this small developer couldn't afford to have happen. I was able to step in late in the game and secure a few points, landing the contract for their next two games.

Managing Outside Talent Shouldn't Be a Mystery

At some companies, there tends to be a stigma attached to third-party contractors. To some, it seems a highly unnatural act to search beyond the company walls after you've spent a tremendous amount of time and effort collecting and nurturing your own talent. As you might have guessed, I take a different attitude. I've found that artists are more creative in working environments that they have designed for their own purposes. Not keeping to a nine-to-five schedule actually lets them budget their own time and work when they are at their best. Their happiness and security can be heard in their much-inspired work, and any game could benefit from this passion.

When it comes down to it, working with a contracted sound designer is not so different from interacting with any of the full-time developers on your staff. Graphic artists, programmers, composers, actors, and voice talent are all looking to you for the proper motivation and, though the sound designers are not immediately within the corporate view, they respond to the same proper, positive stimulus.

The key points mentioned here, when focused on, can help you achieve fantastic work from a sound design contractor. Even though there are no hard and fast rules, secret formulas, or prescribed methods for creating the consummate assemblage of game sound effects, it can happen. It takes fluid communication and a firm vision from the development team coupled with a sound designer who shows no bounds to their creativity and patience. Together we can take on the gaming world and keep them lining up at the stores. ■

The Audio Development Agreement

Tommy Tallarico is making life easier for both producers and sound designers by trying to standardize sound contracts and the often unpredictable contract negotiation process. With more than 125 games to his credit, he has plenty of experience in this phase of deal-making and after paying lawyers large sums of money to draw up the paperwork, he's still willing to share. At the 1999 Game Developers Conference, he gladly handed out copies of the agreement to anyone interested. Hundreds of people took advantage of his generosity.

This contract, available for download from the *Game Developer* web site (http://www.gdmag.com), is a revised copy that pertains to the sound designer and the created effects. Many points can be added or taken out as needed during negotiations, since both parties are attempting to have their best interests represented. Whether you're a producer or a sound designer, I recommend you download this file and read it over closely so that you get a sense for what the top talent in the field is asking for. Tallarico has granted everyone permission to use it as is, or merely as a guideline for sound design deals.

FOR FURTHER INFO

Places to Find Third-Party Sound Designers:

Gamasutra.com
http://www.gamasutra.com

Happy Puppy
http://www.happypuppy.com/biz/index.html

Dungeon Crawl
http://developer.dungeon-crawl.com

Black Sheep Journal
http://members.aol.com/blaksheepj

GameDev.Net
http://www.gamedev.net/info/about

Using Bitmaps for Automatic Generation of Large-Scale Terrain Models

by Kai Martin

Players today demand a rich game experience with larger worlds to explore, more interesting things to do, and higher degrees of realism with each new title that ships. The problem is that game development schedules and budgets cannot keep pace with consumer demand for new feature sets. So, how do we make larger, more interesting worlds without blowing milestones and spending large sums of money? Simply put, we must have some of our game data generated automatically for us.

For example, suppose you're developing an online massively-multiplayer game with an enormous amount of polygonal terrain (hundreds of thousands of screens' worth of in-game scenes) for thousands of players to exist in and interact upon. In addition to that (just to make your life more difficult), this terrain model must conform to a loose, preexisting map specification (in other words, the general map layout and major landmark locations are known relative to each other, but there is no concrete data set describing the terrain, such as satellite imagery). This constraint eliminates the possibility of using any truly automatic terrain generation algorithm (such as fractal terrain generation). Meticulous construction of the terrain model by artists' hands is completely out of the question. No group of artists assigned this arduous task would be able to produce the desired result within the budget constraints; it will either cost a prohibitive amount of money, or take more time than is allotted for the development of the product. So you're left searching for some kind of middle ground between these two extremes.

Using manageably-sized, artist-generated bitmaps, combined with some clever image processing techniques, you can create a desirable terrain model. The techniques I describe in this article don't eliminate artist or world-builder involvement from the creation process — these techniques only create a model that is very close to completion in a relatively short amount of time. Once generated, the terrain

Kai Martin is a programmer for Sierra Online, where he is currently working on, among other things, nearly all aspects of automatically generating an enormous landscape for MIDDLE-EARTH, a massively-multiplayer online role-playing game based on the works of J. R. R. Tolkien. Questions, comments, and miscellaneous musings can be directed to him at kai.martin@sierra.com.

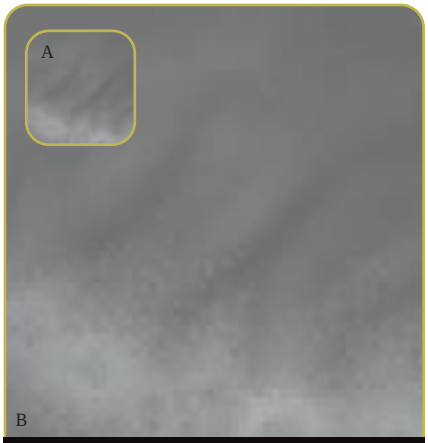
must be fine tuned by an artist or level designer to add the aesthetically pleasing final touches.

There are several other advantages to using bitmaps for your terrain modeling. First and foremost, the tools for manipulating images (such as Photoshop) are extremely well developed and well known by a majority of artists. Second, the techniques I'm about to discuss will lower the ratio of time spent generating the terrain images to the amount of in-game data you can generate from them. Finally, this technique lets you view the layout of the entire world within a fairly small area — the bitmaps we will use are fairly manageable and allow you to view the entire image at once on a typical monitor.

FIGURES 1A-C. *A. Original terrain bitmap. B. Scaled terrain bitmap. C. Scaled and smoothed terrain bitmap.*

It's assumed that the terrain model desired is a textured 3D polygonal mesh, with the vertices lying upon a regularly-spaced rectangular grid. The terrain's *z*-values (the "up" vector) are taken from a two-dimensional array of values called a height field. The terrain textures are also set according to a two-dimensional array of values, where different values denote specific types of terrain. This helps specify a texture to use for a particular cell in the terrain grid. Thus, two bitmaps are required to generate all the data needed to create terrain. In the examples provided in Figures 1a–c and 2a–b, the height and terrain data bitmaps are 8-bit grayscale and palletized color images, respectively.

Note that there are several methods by which you can use this terrain data to create a system of connecting tiles. For example, you could view each terrain value as an individual tile, or view each value as a tile vertex, and so on. However, this article is strictly concerned with generating the needed data. Showing you what to do with the data once it has been generated is beyond the scope of this article.



FIGURES 2A AND 2B. *A. Original elevation bitmap. B. Scaled elevation bitmap.*

Bitmap Representation of Terrain and Elevation Data

Translating bitmap values into usable data is straightforward, assuming you can read the file format of the bitmap. For a given entry (x, y) in a height data bitmap, a corresponding value in the height field can be calculated by taking the value in the bitmap and multiplying it by some scalar: heightField (x, y) = bitmap $(x, y) \cdot$ scaleZ. Using the terrain bitmap is even easier. Since the bitmap is 8-bit, simply set the value (palette index) at any given point (x, y) in the image to a predetermined terrain type. If more than 256 terrain types are needed, you can use the RGB values of a 24-bit image for terrain type indexing instead.

While the goal is to use a large data set to generate a terrain model, it is unlikely that a 1:1 mapping of bitmap values to height and terrain values will yield a large enough data set for very large game worlds. Thus, you probably will have to "scale up" the bitmaps some way in order to generate sufficient amounts of data.

There are two easy ways to scale the data gathered from the height bitmap. The methods rely on a two-dimensional scale vector (scaleX, scaleY) to do the work. The scale vector is created based on the ratio of the size of the bitmap to the size of the terrain model one wishes to create from the bitmap.

The first method takes each pixel (x, y) in the height bitmap and duplicates the pixel value bitmap (x, y) inside a rectangular area of pixels scaleX · scaleY in size, in which the upper left corner of the rectangle equals $(x \cdot$ scaleX, $y \cdot$ scaleY). Empirically, the data becomes "pixellated," as though the bitmap is viewed at a higher zoom level. The terrain data is scaled in this way, as well.

The second method of scaling the height data treats the bitmap values as points on an arbitrarily large surface, or as control points used to generate such a surface parametrically, in which each value in the height bitmap is a discrete sample from this surface. For a given entry (x, y) in one's height-data bitmap, a corresponding value in the height field can be calculated using the following mapping function:

$$[x, y, \text{bitmap}(x, y)] = [\text{scaleX} \cdot x, \text{scaleY} \cdot y, \text{scaleZ} \cdot \text{bitmap}(x, y)]$$

All we're doing is taking a point in the height data bitmap and multiplying it by a scale vector of (scaleX, scaleY, scaleZ).

Now that the amount of raw data needed to create the full size terrain model has been generated, one might notice (see Figures 1a–c, 2a–b, and 3a) that scaling the

bitmaps has created some rather harsh and unwanted artifacts in the final images. To correct these artifacts, let's use some basic filtering techniques from the field of image processing.

Basic Image Processing and Elevation Smoothing

Many image processing operations can be modeled as a linear system:

$$\text{Input}, f(x,y) \rightarrow \left[\text{linear system}, g(x,y)\right] \rightarrow \text{Output}, h(x,y)$$

where $f(x, y)$ and $h(x, y)$ are the input and output images, respectively, and $g(x, y)$ is the system's impulse response. To put it another way, $g(x, y)$ is the operation upon $f(x, y)$ that creates $h(x, y)$. For such a system, the output $h(x, y)$ is the convolution of $f(x, y)$ with the impulse response $g(x, y)$, defined in discrete terms, for an $N \times M$ image, as:

$$\begin{aligned} h[i,j] &= f[i,j] \cdot g[i,j] \\ &= \sum_{k=1}^{N} \sum_{l=1}^{M} f[k,l] \cdot g[i-k, j-l] \end{aligned}$$

If f and h are images, convolution becomes the computation of weighted sums of the image pixels. This computation is performed using an arbitrarily-sized square table of values called a convolution mask. This computation is what performs the actual filtering.

One of the simplest filters is implemented by a local averaging operation where the value of each pixel is replaced by the average of all the values in its local neighborhood, as determined by the size of our convolution mask. For example, taking a 3×3 neighborhood about the pixel (i, j) yields:

$$h[i,j] = \frac{1}{9} \sum_{k=i-1}^{i+1} \sum_{k=j-1}^{j+1} f[k,l]$$

If $g[i, j] = 1/9$ for every $[i, j]$ in the convolution mask, the convolution operation reduces to a local averaging of the 3×3 grid of pixels centered on pixel $[i, j]$. Notice that for the 9 pixels involved in the operation, the sum of the weights is equal to 1 ($9 \times 1/9 = 1$). When an $N \times N$ convolution mask is used as an averaging filter, the size of N controls the amount of filtering. As N becomes larger, the image noise is reduced, but you also lose more image detail. So there's a trade-off in choosing a particular size N, and choosing the size of your convolution mask will depend on the amount of filtering you need and level of detail your final image requires. An example of an average filter applied to a height field is shown in Figure 3b.

A Gaussian filter is similar to the averaging filter. In the Gaussian filter, the values in the mask are chosen according to the shape of a Gaussian function. For reference, a zero-mean Gaussian function in one dimension is:

$$G(x) = e^{-x^2/2\sigma^2}$$

where the Gaussian spread parameter determines the width of the Gaussian. For image processing, a two-dimensional zero-mean discrete Gaussian function,

$$G[i,j] = e^{-\left(i^2+j^2\right)/2\sigma^2}$$

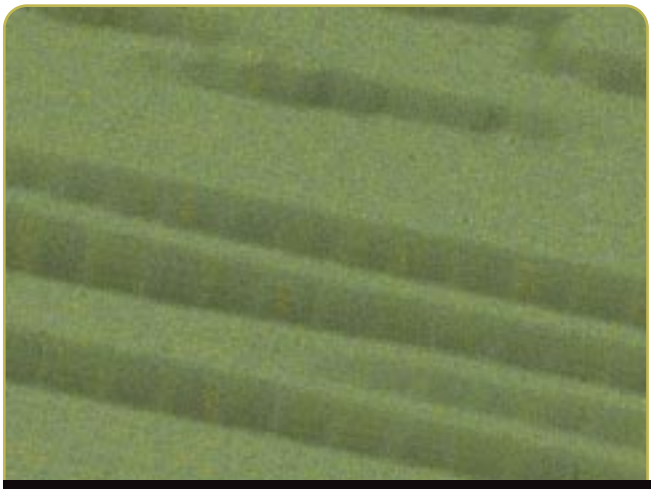


FIGURE 3A. *Scaled, unfiltered height data.*

FIGURE 3B. *Scaled height data, filtered using the averaging filter.*

FIGURE 3C. *Scaled height data, filtered using a Gaussian filter.*

is used as a smoothing filter. The Gaussian filter has a few properties that make it particularly useful for smoothing purposes.

First, the Gaussian function is rotationally symmetric. In other words, the function does not favor any particular direction when it smooths, which is particularly useful when the areas needing smoothing are oriented in an arbitrary direction (not known in advance), and there is no reason to smooth in any specific direction.

Second, the Gaussian function has a single lobe, which means that the Gaussian filter replaces each pixel with a weighted average of the neighboring pixels around it (like the averaging filter), such that a pixel's weight decreases monotonically with distance from the central pixel.

TABLE 1. *Grid of values.*

[i, j]	-3	-2	-1	0	1	2	3
-3	.105	.197	.287	.325	.287	.197	.105
-2	.197	.368	.535	.607	.535	.368	.197
-1	.287	.535	.607	.779	.607	.535	.287
0	.325	.607	.779	1.000	.779	.607	.325
1	.287	.535	.607	.779	.607	.535	.287
2	.197	.368	.535	.607	.535	.368	.197
3	.105	.197	.287	.325	.287	.197	.105

TABLE 2. *Resulting table of values.*

[i, j]	-3	-2	-1	0	1	2	3
-3	1	2	3	3	3	2	1
-2	2	4	5	6	5	4	2
-1	3	5	6	7	6	5	3
0	3	6	7	10	7	6	3
1	3	5	6	7	6	5	3
2	2	4	5	6	5	4	2
3	1	2	3	3	3	2	1

The filter centers on one pixel (i, j). This pixel is modified by: 1) Multiplying each surrounding pixel, including the center pixel by its respective filter weight, and adding the resulting products together. 2) Dividing the sum from step one by the sum of the filter weights. This is the new value for pixel (i,j). This allows local features in the height bitmap to remain in the filtered image. Finally, the width (and thus the degree of smoothing) is linked directly to σ, so that as σ increases, so does the degree of smoothing. One can control this parameter to achieve a balance between the amount of smoothing and blurring in the final image.

There are a couple of techniques that one can employ to determine what kind of Gaussian filter to use. If the filter is being calculated directly from the discrete Gaussian distribution

$$G[i,j] = ce^{-\left(i^2+j^2\right)/2\sigma^2}$$

where c is a normalizing constant, the equation can be rewritten as

$$\frac{G[i,j]}{c} = e^{-\left(i^2+j^2\right)/2\sigma^2}$$

Once a value for σ^2 is chosen, the function can be evaluated over the $N \times N$ area desired for the mask. For example, choosing $\sigma = 2$ and $N = 7$, the above equation yields the grid of values in Table 1. However, if integer values are desired inside the mask, you can divide every value inside the mask by the value at one of the corners in the array (the smallest value in the mask). With this completed, and assuming the values are rounded appropriately, a table is created like that shown in Table 2.

Notice that the sum of all the weights contained in the above Gaussian masks do not equal one. Thus, the result given from convolving a given section of the image by the mask should be divided by the sum of the weights contained in the mask. This ensures that the mask does not affect regions of uniform intensity. An example of the Gaussian filter shown above applied to a height field is seen in Figure 3c.

Another useful aspect of the Gaussian function is the fact that it is a separable function. In other words, a two-dimensional Gaussian convolution can be obtained by convolving the bitmap with a one-dimensional Gaussian and then convolving the result with the same one-dimensional Gaussian–oriented orthogonal to the Gaussian used in the first step. Therefore, another way to create a Gaussian filter is to approximate it by using the coefficients of the binomial expansion (you might remember the binomial series from calculus, where it was used to estimate integrals and roots of numbers):

$$(1+x)^n = \binom{n}{0} + \binom{n}{1}x + \binom{n}{2}x^2 + \cdots + \binom{n}{n}x^n$$

In other words, use row n from Pascal's triangle (Figure 4) as the values for your Gaussian filter. For example, a five-point approximation of a Gaussian filter is:

1 4 6 4 1

This corresponds to the fifth row in Pascal's triangle, as shown in Figure 4. This method works for filter sizes up to around $n = 10$. For larger filters, the binomial coefficients become too large for most images. Of course, if floating-point values can be used, you could always normalize the row by the largest value.

Depending on how much you scaled the original height bitmap, the above smoothing methods should produce satisfactory results. However, for higher amounts of scaling, the amount of smoothing needed (provided by either of the methods above) might be too high to produce a smooth height bitmap (depending on what kind of terrain model will be satisfactory). In doing so, there is a trade-off between losing local detail from the original height bitmap (since smoothing reduces noise by spreading it over a larger area, making it more difuse) and generating more terrain data from a bitmap of given size.

Smoothing Using Curved Surfaces

With images that have a large amount of height variation, such as a terrain that goes from a valley at sea level to a mountain peak, the amount of smoothing needed to produce a satisfactory terrain model would be so large

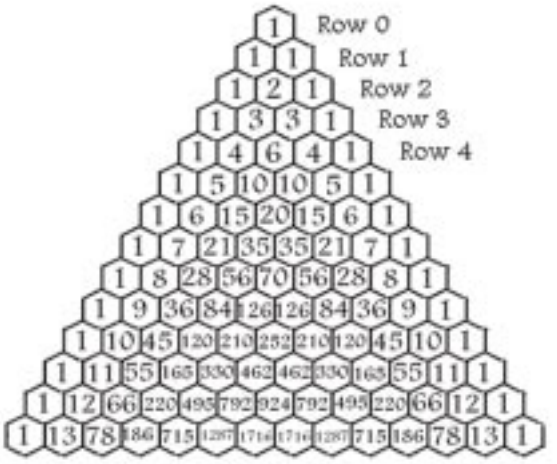


FIGURE 4. *Pascal's triangle. At the tip is the number 1, which makes up the zeroth row. The first row (1 & 1) contains two 1s, both formed by adding the two numbers above them to the left and the right, in this case 1 and 0 (all numbers outside the triangle are zeros). Do the same to create the second row: 0 + 1 = 1; 1 + 1 = 2; 1 + 0 = 1. And the third: 0 + 1 = 1; 1 + 2 = 3; 2 + 1 = 3; 1 + 0 = 1. In this way, the rows of the triangle go on infinitely. A number in the triangle can also be found by n Choose r (nCr) where n is the number of the row and r is the element in that row. For example, in Row 3, 1 is the zeroth element, 3 is element number 1, the next 3 is the second element, and the last 1 is the third element. The formula for nCr is shown below.*

$$\frac{n!}{r!(n-r)!}$$

The ! symbol means factorial, or the preceding number multiplied by all the positive integers that are smaller than the number. 5! = 5 x 4 x 3 x 2 x 1 = 120.

that a great amount of detail would be lost in the process — your mountains might suddenly turn into rolling hills. So a different approach for these images must be used. A more obvious yet slightly more complicated method is to find some form of surface representation from the initial set of data points. This can be a surface either shaped by or interpolating through these points. There are many ways to do this, but this discussion will be limited to creating a uniform cubic B-spline surface to achieve our goal.

Introduction to B-Splines

I will assume that you have some basic knowledge of parametric curved surfaces, such as Bézier curves and surfaces. If not, Brian Sharp's articles on Bézier curves and surfaces ("Implementing Curved Surface Geometry," June 1999, and "Optimizing Curved Surface Geometry," July 1999) are a very good introduction.

You may recall that four control points define a cubic Bézier curve and that 16 control points define a cubic Bézier surface. In general, a degree n Bézier curve is defined by $n + 1$ control points, and a degree n Bézier surface is defined by $(n + 1)^2$. However, the desired final set of elevation points will be much larger than a 4×4 grid. Therefore, one will need to use either a much larger degree Bézier surface, or employ another approach that will remain a third degree surface that allows any number of points to define it.

This is where the cubic B-spline comes in. In simple terms, a B-spline of degree n can be thought of as a composite curve made up of several curve segments, each also of degree n, with each curve segment defined by $n + 1$ control points. In this case, each curve segment is a third degree curve defined by four control points. An important feature of the B-spline is what's known as "C^2 continuity." This means that at any point on the curve, the second derivative will exist (means that no sharp points will exist anywhere on the curve — a nice property to have). To define the B-spline as a whole, if we have $m + 1$ control points, then we have $m - 2$ curve segments. Let a given curve segment Q_i be defined over the interval $0 \le u \le 1$ by basis functions $B_k(u)(k = 0,\ldots,3)$ and control points p_i, p_{i+1}, p_{i+2}, p_{i+3} as follows:

$$Q_i(u) = \sum_{k=0}^{3} p_{i+k} B_k(u)$$

This should look familiar, since it's very reminiscent of how a Bézier curve is defined. For the sake of brevity, here are the basis functions for a cubic B-spline (if you'd like more information how the functions are actually derived, please see the References section at the end of this article):

$$B_o(u) = \frac{(1+u)^3}{6}$$

$$B_1(u) = \frac{(3u^3 - 6u^2 + 4)}{6}$$

$$B_2(u) = \frac{(-3u^3 + 3u^2 + 3u + 4)}{6}$$

$$B_3(u) = \frac{u^3}{6}$$

Since these segments are connected together to create one large curve, it makes sense to have a parameterization of the entire curve in terms of one parameter U, instead of having just a parameter u for every curve segment. Globally, U is defined over $[0, m - 2]$, and u defined (as one would expect) over $[0, 1]$. The parameter u for any given curve segment i is given by $U - i$.

Except for the end points of the B-spline curve, each control point (in the case of a cubic B-spline) influences four curve segments, that is, control point p_i influences curve segments Q_{i-3}, Q_{i-2}, Q_{i-1}, and Q_i. The influence of each control point p_i over the curve at some global parameter value U is "collected" into one global function (called a blending function), $N_i(U)$.

Think of the blending function as the sum of the basis functions (which is similar to the basis functions used when evaluating Bézier curves) for any given point on the curve. In formal terms, the B-spline curve $Q(U)$ is defined by:

$$Q(U) = \sum_{i=0}^{n} p_i N_i(U)$$

where

$$N_i(U) = \begin{cases} B_3(u_0) & i-3 \le U < i-2 \\ B_2(u_1) & i-2 \le U < i-1 \\ B_1(u_2) & i-1 \le U < i \\ B_0(u_3) & i \le U < i+1 \end{cases}$$

$$u_j = U - i + 3 - j \quad (j = 0, \ldots, 3)$$

The global function $N_i(U)$ takes the global parameter U and converts it to the appropriate local parameter u_j for each curve segment involved.

As mentioned earlier, this type of B-spline is a *uniform* B-spline, which means that the curve consists of curve segments between endpoints that are spaced equally apart (there are other types of B-splines that do not have the ends equally spaced, but we won't worry about those here, since we're dealing with a regular grid of control points). In our case, these ends are located at [0, 1, …, m] of U for m number of control points. These ends will be referred to as knots (to be consistent with any other reference one may find on B-splines). Thus, the spline curve becomes a collection of the knot intervals $t_0 < t_1 < \cdots < t_m$. Also, the order k (the degree + 1) of a B-spline can vary, but for simplicity's sake, we will keep the order of our spline constant at $k = 3$. Now we can recursively define a blending function $N_{i,k}(u)$ of degree k over the knot range $[t_i, t_i + k]$ as follows (otherwise known as the Cox de Boor algorithm):

$$N_{i,o}(u) = \begin{cases} 1 & \text{if } t_i \le u \le t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$N_{i,k}(u) = \frac{(u - t_i)N_{i,k-1}(u)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - u)N_{i+1,k-1}(u)}{t_{i+k} - t_{i+1}}$$

Now that we have the definition of a B-spline curve, how do we define a B-spline surface? Informally put, any point (u, v) on the surface is calculated by multiplying two separate curves together. Formally, let a point (u, v) on a cubic B-spline surface S defined by a grid of control points $p_{i,j}(i = 0, \ldots, n; j = 0, \ldots, m)$, and blending functions $N_{i,k}(u)$ and $M_{j,k}(v)$

$$S(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} p_{i,j} N_{i,k}(u) M_{j,k}(v)$$

Now that we have basic knowledge of B-splines, applying what we have learned to create our desired amount of elevation data should be relatively easy. The initial set of elevation points should be used as control points for the final surface. Next, determine the dimensions the final elevation data should satisfy. Let our final elevation data set be s values wide by t values tall. When evaluating points on the surface, one needs to know the change in $u(du)$ and change in $v(dv)$ from one point to the next, which is defined by:

$$du = \frac{n}{s}, \quad dv = \frac{m}{t}$$

The code to do all of this is available on the *Game Developer* web site, http://www.gdmag.com.

Terrain Smoothing

In smoothing the terrain bitmap, neither the straightforward smoothing algorithms described above nor any other image processing technique used for scaling or smoothing images can be applied. All of these methods have potential to introduce new color values into the final image, and only the terrain values contained in the original terrain bitmap can be present in the final bitmap. In this smoothing method, another $n \times n$ array of values centered on a pixel $g[i, j]$ in an image g will be used to calculate the value of the pixel $h[i, j]$ in the final image h. However, instead of using an equation or other means independent of the values contained in the source bitmap, the values in the $n \times n$ convolution mask will be taken directly from source bitmap (specifically, the $n \times n$ neighborhood surrounding pixel $g[i, j]$). Next, a histogram (an "inventory" of all the different values contained in the specific $n \times n$ area of the source bitmap) of the $n \times n$ array is calculated. From this histogram, the value that is most frequently occurring in the $n \times n$ region surrounding the pixel $g[i, j]$ shall be the value of the pixel $h[i, j]$.

Conclusion

The convenience of using bitmaps for generating game data can extend beyond just polygonal terrain generation. Other examples could be world object placement (for trees and other vegetation), non-player character (NPC) placement (perhaps for a real-time strategy game where hundreds of units need to be placed for a given scenario), setting paths for NPCs to follow, or providing any addition information about the terrain (for example, setting "off-limits" areas for certain player characters or NPCs). The number of things that an image can represent is virtually limitless. So, before considering creating your own custom tools for generating game data, make sure that you're not simply reinventing the wheel by wanting to provide something that a bitmap could provide just as well. ■

REFERENCES

Image Filtering

- Gonzalez, Rafael C., Richard E. Woods, and Ralph Gonzalez. *Digital Image Processing*. New York: Addison-Wesley, 1992.
- Jain, Ramesh, and others. *Machine Vision*. New York: McGraw-Hill College Division, 1995.

Curved Surfaces and Surface Interpolation

- Rogers, David F. *Mathematical Elements for Computer Graphics*. New York: McGraw-Hill College Division, 1989.
- Watt, Alan and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. New York: Addison-Wesley, 1992.

POSTMORTEM

0 age DESCENT 3

by Jason Leighton and Craig Derrick

DESCENT 3's creation was a long and arduous task that was both a joy and a pain. Exciting technology coupled with inconsistent design and almost nonexistent management had all members of the team exhausted by the end of our 31-month development cycle. When the game finally did ship however, we knew we had a winner.

Developed by Parallax Software and published by Interplay Productions, DESCENT was released in 1995 and took the world by storm with the first first-person shooter offering 360 degrees of movement. No longer were players constrained just to walking around a 2D world — now they had complete freedom of movement in a true 3D space.

Jason Leighton likes to kill time by complaining about the horrid Michigan weather cycles. When it is actually nice outside, he complains anyway. To hear his weather woes, or if you just want to talk about game programming, write to jason@outrage.com. Your e-mail is important to him, but may be monitored for quality assurance purposes.

When Craig Derrick isn't off buying DVDs, he's often writing to Jason Leighton about his weather woes and trying to convince him that all he really needs to do is to stop programming and go outside. If you need a motivational speaker or if you have a problem and no one else can help, write to craig@outrage.com.

Players were able to fly their Pyro ship in any disorienting direction — up, down, and everywhere — and top became bottom, bottom became up, as players plummeted down never-ending tunnels blasting mechanical robots and saving imprisoned miners.

This innovation in action gaming was immediately successful and garnered The Academy of Interactive Arts and Sciences Best Title of 1995 and Best Computer Game of 1995. And *PC Gamer* voted it Best Action Game in the World. One year later, the sequel DESCENT II was released, which added another 30 levels to the mix and improved the game's AI, thanks to the addition of the Guide-Bot and Thief-Bot. The game ended with a cliffhanger cinematic that almost certainly guaranteed another sequel. That's where the DESCENT 3 project began.

Who the Hell Is Outrage?

DESCENT was developed by a small group of programmers and artists at Parallax Software in Champaign, Ill., headed by Mike Kulas and Matt Toschlog. After the successful completion of DESCENT, Toschlog moved to Ann Arbor, Mich., and established a second office for Parallax Software. He took three designers with him and hired two additional programmers. Both offices then began to work simultaneously on DESCENT II. While DESCENT II was deemed a successful project, the process of trying to get teams located in two distant offices to work effectively together took a heavy toll on both teams. It was at that time that Matt and Mike decided that each office should work on separate titles and eventually become separate companies. Thus, Outrage Entertainment and Volition Inc. were born.

It was another eight months before the production of DESCENT 3 began. After the release of DESCENT II, Outrage immediately began work on THE INFINITE ABYSS (a Windows 95 version of DESCENT II, along with a new level add-on pack entitled THE VERTIGO SERIES). Meanwhile, Volition concentrated on development of FREESPACE. It wasn't until after the release of THE INFINITE ABYSS and DESCENT MAXIMUM (the Playstation version of DESCENT II) that the developers here at Outrage began focusing all our energy on the design and development of DESCENT 3.

Another Sequel And Why DESCENT 3?

DESCENT I AND II had the makings of a franchise for Interplay, and with any franchise, successful or otherwise, sequels are sure to follow. Technology had taken a big leap in the year and a half that DESCENT had come out: notably Windows 95 and hardware-accelerated 3D. It was easy to see how DESCENT 3 could be dramatically improved over its predecessors.

By the fall of 1996, we began to compile a list of features that we would like to see in DESCENT 3. By November we had created a design document detailing the new features that would be implemented for DESCENT 3 and submitted it to Interplay for approval.

The initial design and programming work on DESCENT 3 began in December 1996. Some of the team had just completed work on DESCENT II – THE INFINITE ABYSS and DESCENT MAXIMUM for Playstation, while others were involved with research and development for DESCENT 3, where they learned about new tools and technologies. We were excited about taking the DESCENT franchise to the next level and eager to begin, but little did we know that our over-eagerness would impair the game's development.

Initial conceptual drawing of the Phoenix Interceptor.

About six months after starting development, we stepped back and took a long hard look at what we had and where we were going. Originally, it was deemed that DESCENT 3 would have both a software and hardware renderer. After checking out the competition, it was apparent that, if we wanted to be visually stunning (and maintain interactive frame rates), we would either have to scale back our technology design or go with a hardware renderer only. We chose the latter. In retrospect this was a good decision, but it was unfortunate that we had to make it six months into the development, since many of the tools and software rendering technology were already developed.

At this point, not only did we decide to go with a hardware-only renderer, we decided to scrap the engine that we had been developing. The engine we had going was an

DESCENT 3

Outrage Entertainment
Ann Arbor, Mich.
(734) 663-9120
http://www.outrage.com
Release date: June 1999
Intended platform: Windows 95/NT/98
Project budget: $2 million
Project length: 31 months
Team size: 19
Critical development hardware: Intel Pentiums and AMD K-6 II processors. Each machine was equipped with a hardware accelerator and at least 64MB of RAM.
Critical development software: Photoshop, 3D Studio Max, Lightwave, Microsoft Visual C++ 4/5/6; rendering APIs used include Sourcesafe, OpenGL, Direct 3D, and Glide; Direct Sound, Aureal, and EAX were used for the game audio.

enhanced segment engine — a portal engine that used six-sided deformed cubes to represent geometry. Essentially, it was the same technology used in DESCENT II, with some additional features thrown in for improved geometry modeling. If we had stuck with this engine until the game shipped, we would have been way behind the technology curve with respect to our competition. Instead, we went with a "room"-based engine, which allows designers to create just about any geometrical area within a 3D modeling program, such as 3D Studio Max or Lightwave.

Shortly thereafter, the terrain engine was developed. We were seriously considering the idea of creating outdoor areas for DESCENT 3, but we worried about the high polygon count associated with such a large rendering distance. Fortunately, we used a good level-of-detail (LOD) algorithm to combat the frame-rate problems. Unfortunately, the decision to include terrain would adversely affect the overall design in ways that we couldn't possibly have foreseen, such as the scale of the terrain dictating how fast the ship appeared to move while outside.

For the next 18 months, work continued on DESCENT 3 at a frantic pace. We were learning how to use our custom in-house tool, D3Edit, so in the process we ended up creating and then throwing out an incredible amount of our content — what looked cool one month looked dated the next. This was largely due to the fact that we were developing a cutting-edge engine at the same time we were trying to design the game itself — a pitfall many developers have fallen victim to. Unfortunately, throwing out so much work also cost our team a lot in terms of our morale. What we should have done is freeze the design of the engine about a year before the product shipped and then worked on the game. Unfortunately, in our lust for sexy technology, we just couldn't do that.

When we finally did ship, we were exhausted in a variety of ways. Working on the same game for two and a half years is emotionally depleting, to say the least. Although we knew the game was cool, we didn't know how the public (or reviewers) would receive it. Thankfully, it turned out that our fears were unfounded — DESCENT 3 has received incredibly good scores from a variety of sources, including print magazines and gaming web sites.

What Went Right

1. WORKING ON A SEQUEL. Working on any sequel, whether a game or movie, has its ups and downs, and DESCENT 3 was no exception. Much of the joy of working on a sequel comes from the fact that you can improve on an already established title, and in many cases, add features that were previously impossible to do.

Knowing your target market is essential to making a successful sequel. You must not deliver a product that completely turns away your core audience, and yet enough of it must be new in order to persuade them to purchase it. The four long years between DESCENT II and DESCENT 3 convinced us that new technology alone warranted a new product, but we didn't want just to copy the previous version. DESCENT 3 had to retain all of the elements that made its predecessors big hits and yet, creatively, be different. And therein lies the paradox of making a sequel.

Starting with our old games' design document and features list, we cataloged elements that the team liked and disliked about the previous games. Every item was scrutinized and broken down into specific lists, such as art, weapons, sounds, user interface, and so on, to determine what part of the feature needed to be changed and what needed to be left alone. Developers sometimes do this with their competition's products when beginning development of a similar game, but nothing beats being able to do it with your own game because you have an intimacy with the product that outsiders are not privy to.

Conceptual drawing of the hulking Juggernaught robot.

2. USING A HARDWARE-ACCELERATED 3D ENGINE. When development of DESCENT 3 began in November 1996, hardware accelerators (specifically 3dfx's Voodoo 1) had just come out. Our initial design document called for DESCENT 3 to ship with a hardware and software renderer, but as our aspirations for the graphics engine grew, so did the need for hardware acceleration. Complex geometric rooms, robot enemies having nearly twice the amount of polygons and animation states compared to our previous games, complex outdoor terrain, and the whiz-bang effects expected in today's games all necessitated hardware acceleration. With all these features in the game and running at abysmal frame rates with our software renderer, it was decided, reluctantly, that we would release as a hardware-only game.

As development wore on, technology advanced and accelerators became faster, cheaper, and more popular with each passing year. It seemed that games were coming out every week that sported a hardware accelerator mode or patch, but up to this point, nothing had come out that was hardware-only. We knew just by looking at our progress on the game under acceleration that we had a beautiful looking game with all the latest technologies — but would anyone actually be able to play it?

Our Christmas deadline came and went, but in retrospect I feel it was for the better. If you didn't have a hardware accelerator before Christmas 1998, chances are you did have it later. With the implementation of Direct3D, OpenGL, and Glide, DESCENT 3 was capable of running on just about every video card available when it was released. Because we took a chance on technology, believed

The new and improved Thief-Bot.

Sparky, one of the more than 30 new robots in DESCENT 3.

in our product, and slipped a bit, DESCENT 3 looks, feels, and plays like a next-generation DESCENT. And that's all we really wanted.

3. OUT WITH THE OLD ENGINE, IN WITH THE NEW. As mentioned earlier, the initial plans for DESCENT 3's graphics engine were to include both a software and hardware renderer. The engine itself was to be a heavily modified version of the segment (cube-based) engine used for DESCENT I and II, meaning that all geometry had to be sculpted by connecting one deformable cube to another. While this engine supported some interesting geometry, it just couldn't handle the complexity we had in mind for DESCENT 3's levels.

So, six months into development we started over on the engine, and this time we aimed higher. We set our sights on creating what is now the Fusion Engine. This engine would actually be two separate engines — one for internal settings and one for outdoor terrains — that would work together seamlessly. The internal "room" engine allows designers to model almost any type of complex geometry in a program such as 3D Studio Max and import it directly into our game editor. Once imported, designers can modify the geometry, texture it, place objects, and light it. Designers could then take the individual rooms and join them together, creating a portalized world for the player to fly through.

The terrain engine actually began as a prototype for another game that Jason was interested in developing. Unfortunately, Bungie's MYTH beat us to the idea, but the terrain technology was solid enough to be incorporated into DESCENT 3. It was based on a great paper by Peter Lindstrom and colleagues entitled *Real-Time, Continuous Level of Detail Rendering of Height Fields* (from Siggraph 1996 Computer Graphics Proceedings, Addison Wesley, 1996). Of course, it was bastardized heavily to fit the needs of DESCENT 3, but the overall concept was the same — create more polygonal detail as you get closer to the ground and take away polygons when you are farther away. After implementing the real-time LOD technology, our frame rates quadrupled.

The outdoor engine gave designers the ability to create an internal structure and its outside shell (an external room with the normals flipped), and place it anywhere on the terrain. This let us create seamless transitions between a structure within the level to an outdoor section, with absolutely no load times whatsoever. For the first time in DESCENT, players could actually leave the mine. When players cross the portal that leads from inside to outside, the game code would switch collision detection, rendering, and so on, to use the terrain engine.

4. INCREDIBLE TECHNOLOGY. One of our biggest goals in developing DESCENT 3 was to bring the game engine up to date. This included graphics, AI, sound, and multiplayer. I think we hit the mark. DESCENT 3 includes just about every whiz-bang graphical feature there is, the AI is very smart for an action game, and the multiplayer plays pretty well even over lagged connections. Even though we're not in the first-person-shooter genre — a genre that is judged by its graphics and networking technology (some say at the cost of game play) — we compete favorably with the offerings in that arena.

5. GREAT MULTIPLAYER. We knew that DESCENT 3 had to have the best multiplayer right out of the box, or people would be disappointed and scream for our heads. Sadly, in this age of release-once, patch-many, there are a lot of games that come out that haven't fully tested their multiplayer aspects, and these systems are full of bugs. Fortunately, DESCENT 3 did not suffer from this. We spent a lot of time testing DESCENT 3 networked games over a variety of conditions, both lagged and unlagged. It was a tiresome process, but in the end, I'm happy we did it.

Another thing we did that showcased our attention to multiplayer was to give a whole slew of options to the player. We had support for IPX, TCP, DirectPlay Modem, and DirectPlay Serial. These options allowed players to connect to games using the protocol best suited for their situation, instead of just offering TCP as a lot of other games do. There were also three network architecture types: D3 client/server, peer-to-peer, and permissible client/server. We did this because we knew we had to support the DESCENT II fans (peer-to-peer), the QUAKE and UNREAL fans (permissible client/server), and try to forge our own path with a new network technology

(D3 client/server). We guessed that permissible client/server would be the most popular model, simply because that was what players were used to. To our surprise, it turned out that D3 client/server was the most frequently used architecture. This architecture changed the way lag was perceived. In games such as QUAKE and UNREAL, there is a noticeable delay between firing the weapon and when the weapon appears on your screen. The reason is that the client asks the server to fire and the server gives the client permission to fire (hence permissible client/server). The D3 client/server is different in that it allows you to fire right away — when you press the trigger, the laser appears immediately. The downside is that you have to lead your opponent by your ping to the server, and sometimes when the laser would hit your opponent on your screen, it wouldn't really hit them on the server. We found this to be less frustrating that the permissible client/server way (which personally drove me crazy), and thankfully our fans agreed.

While providing players with so many options might have cost us development time, I'm confident that we made up for it in the satisfaction that our customers had by being able to customize the game to their liking.

Overall, the development team at Outrage was very energetic to work on DESCENT 3 and their dedication paid off more than once during development. Working on a game as long as we did is always tough going. Because we, too, are game players, it was sometimes tough to be working on something for so long, never quite knowing whether or not the work you were doing would be accepted by your peers within the industry, or more importantly, by consumers and fans of the product. This all changed for us during 1998's E3 convention.

We didn't get confirmation from Interplay that we would be showing the game at E3 until about a month before the show. When the word finally did come, we shifted gears from our production and went full steam towards making the best E3 demo that we could. This would be the time that the industry would get its first glance at our game and we wanted to come out a winner. Fortunately, everything turned out all right. The fans who got a glimpse at DESCENT 3 were very impressed and the comments from the press were overwhelmingly positive.

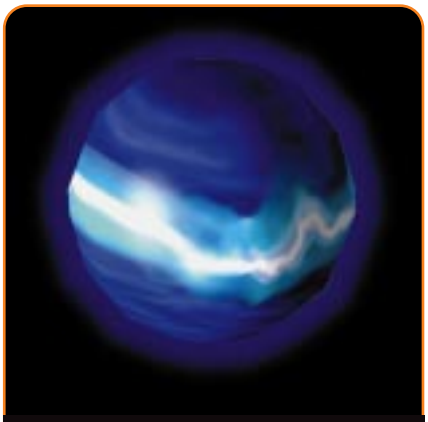
A lot of what kept the team's energy high was the amount of pride that each person had in their own particular domain. The level designers wanted to make the absolute greatest levels ever, the programmers wanted their particular systems to stand out, and the artists wanted the art style of the game to be unique. So this pushed people to do their very best — the atmosphere was almost competitive. There were a couple of times when things got out of hand and egos had to be held in check, but for the most part, the individual team members were allowed to shine without stepping on the toes of a fellow colleague.

What Went Wrong

1. LACK OF DIRECTION OR VISION. The biggest problem on DESCENT 3 was the weak management structure. DESCENT and DESCENT II were developed by small groups that worked closely together (often in the same room), and as we grew to a team of almost twenty people, we didn't introduce enough management to control the process. Even though people had designated titles, there was no real authority attached to those titles. No code reviews, no art reviews, no way of saying, "This is bad and we should be going in a different direction." What we needed was more of a hierarchy and a systematic way to get things done. People would have disagreements that were never settled, and that led to bad feelings among team members. It would have been much better if some sort of hierarchy had been in place to mediate disputes, but unfortunately this didn't materialize during the entire development cycle of the game. For example, if a couple of people had a problem with the way a particular level played, there was no recourse to get that level changed. Bringing up the faults in an individual's level, system, or art would start arguments, and that got us nowhere. Next time we'll know better. An anarchistic development environment is great in theory (total freedom), but you really do need a hierarchy of some sort, a chain of command where people know whom to go to with problems. It was a painful lesson that could have been avoided if we had set up our team to be more like a company and less like a garage band.

2. NO STANDARDIZED LEVEL DESIGN TOOLS. Another major problem on DESCENT 3 was the tools the level designers used to make their levels. Some people used 3D Studio Max, some used Lightwave, and one designer even wrote his own custom modeler from scratch. Due to our somewhat anarchistic development environment, this was seen as O.K., and not too many people complained. However, the different tools led to inconsistent quality across our game levels. One designer would make great geometry that had bad texturing, while another designer would create the opposite, especially if his tool of choice made texturing or geometric modeling difficult. What we should have done is standardized using one tool (probably 3D Studio Max) and made the designers learn how to use it whether or not they were comfortable with it. This would have allowed us to make use of certain features that 3D Studio Max has (such as parametric surfaces) without worrying about whether Lightwave or the custom modeler supported that feature.

Watch out for the dual Fusion attack of the Thresher.



The DESCENT orb, now realized in full 3D.

After assembling their geometry, the designers took their rooms and models, imported them into our custom editor (D3Edit), and tried to glue everything together. Unfortunately, our editor was written by programmers who didn't give enough thought to the user interface — they often designed interfaces that were intuitive to a programmer, but not to a designer. Conversely, a designer would ask for a feature that might take a programmer a long time to code, but then the designer wouldn't use the feature very much. This led to feelings that the designers didn't know what they wanted, and the programmers didn't care enough to make things easier for the designers. It was a vicious circle that didn't get cleared up until the latter third of the project. Even in the shipped game you can tell which levels were made early on and which were made near the end of the production cycle. The later levels are much better looking, have better frame rates, and generally have better scripts.

3. PROGRAMMERS WORKING ON DEDICATED SYSTEMS. Many systems, such as the graphics, AI, and scripting, were written exclusively by one person. This fostered a feeling of pride in a particular system, but it also caught us with our guard down when we found that one system was falling behind schedule. Our AI, for example, was very much behind schedule for the duration of the project because the programmer just had too much to do. This caused a ripple effect that was felt throughout the design of the game. After all, you can't tell how a level is going to play if the robots that you're fighting aren't behaving at all as they should. We couldn't simply add another programmer to that particular system to help out, because by the time the new programmer became familiar with the new system, then his system would fall behind. It might sound as if this was an understaffing problem, but it wasn't. It was more of a case of "design as you go," where someone would suggest a great feature and then the programmers would implement it. Unfortunately, all these excellent features started adding up and taking a tremendous amount of time to implement. It would have been better if there were more programmers on a particular system, or at least ones who were familiar with it, so that if there were concerns with slippage, other people could have been brought aboard to help.

4. WORKING ON A SEQUEL. With sequels come high expectations that can almost never be achieved completely. (Just look at *Star Wars: The Phantom Menace* to see what I'm talking about.) While we aimed high for the entire game, many times trying to meet the varying expectations of our team (more in-level scripting and rendered cinematics), publisher (using more AI and levels), and fans (more of everything) resulted in half-implemented features, such as our in-game cinematics or items that went in untested at the last minute (such as some Guide-Bot functionality).

With a game so big and widely anticipated, I'm not sure if we would have ever been able to manage everyone's expectations. And, one of our biggest problems was that those expectations were controlling the design of the game. As developers, we should have been more confident in our abilities to produce this type of game and should have approached new ideas with a bit of skepticism. While I don't dismiss the valuable role that the fans and our publisher played in the development of DESCENT 3, we were too apt to deviate from our design document when others wanted us to add features to the game.

It's very important to stay true to your design document and know which systems are valuable. Every new feature should be evaluated not only on its value to the game, but also from a production standpoint. When we decided to create in-game cinematics, it was simply to introduce the boss robots. The effect was so impressive that we decided to use the system to establish the player's location in the beginning and ending of each level. This worked out so well (do you see where this is going?) that we used the system to help establish when puzzle elements were completed. This one-time only system was utilized in ways that hadn't been thought of before its implementation, which caused numerous problems.

5. FAST COMPANY GROWTH AND GREEN EMPLOYEES. When we started work on DESCENT 3, Outrage had just eight employees on staff. Since some of the DESCENT II team left Ann Arbor to work at Volition during the project, we literally had to build the team and company at the same time we started production on the game. This resulted in a mixed bag of results ranging from pushing back larger projects or features until later in the project to adding multiple tasks to someone's already full schedule.

One of the results of being understaffed was the decision to contract out most of our 3D animated door modeling to an outside company, Vector Graphics, for completion. Our schedule showed that one animated door took anywhere between three to five person-days to complete. With a design document that dictated more than 30 doors in the game, we would have run out of time. We made the decision to round up all our sketches for doors and hand them over to the very capable hands of Vector Graphics. This allowed our designers to concentrate on their scheduled tasks and then add the doors later as we received them.

What we didn't account for were the problems that came up because our teams worked in different locations. (Somehow we forgot that our company was split for this very reason.) Assembling our development environment and building an editor for Vector Graphics was troublesome, and once we gave the editor to them, they really didn't know how to use it. Our lead designer worked with them almost daily to bring them up to speed and fix any file incompatibilities caused by our constantly evolving editor. This caused our lead designer to fall behind in his schedule, and we had to shuffle around due dates for his specs and level deliv-

erables. As we hired more people, he caught up.

In the end, we hired an additional nine employees, only two of whom actually had any professional game development experience. And while everyone on the team gave the game their full attention, we definitely had issues come up that were the result of inexperience.

A loss of professionalism, maturity, and a standard of conduct was apparent during the development of DESCENT 3. Some of this was due to young employees who came to us directly from school with no professional work experience, while others had trouble dealing with the long hours that come with the job. This, coupled with the lack of a strong management hierarchy, resulted in clashes over direction, seniority, and leadership on the project. More times than not, a person would not follow the direction of the lead simply because of their personal issues with that person. This resulted in the lack of respect for that person and the very responsibilities that came from being the lead.

But our biggest problem was definitely with art direction. Without a dedicated art director on staff, we often second-guessed everyone else's work. Each artist and designer had specialized tasks, and without an art director to make a final decision, art was simply added to the game without any formal approval. In the beginning, we had show-and-tell meetings to show off the latest work from people, but this almost always turned into a forum for criticisms and led to animosity between team members. An art director leading us would have kept our artwork consistent and would have been the final authority on all artwork-related matters.

END GAME DESCENT 3 was an incredibly challenging project, to say the least. The design-as-you-go and design-by-committee aspects were a large part of the problem, and any future Outrage projects will be more diligent in their initial design. We'll make sure that the designers have absolutely the best tools at their disposal and we'll have better management to mediate design disputes. Without these problems during the development of DESCENT 3, I'm fairly confident the whole project would have been a little less painful. ■

by Mike Kelleghan

Who's Eating Your Slice of the Pie?

My boss got a new Ferrari the other day, the same day he laid off about a third of the company. Somehow that smells fishy to me. We've all heard the figures about how much money is coming into this business. Cowles/Simba projects that industry revenues will reach $11.6 billion in 2001. These are real dollars, and I know a lot of the folks who are getting rich in this business. Heck, I develop the games they're getting rich on. I'm not saying I need a new Ferrari every year, but I'm pretty sure I'm not getting my share when it takes me 15 years to save up for a new pickup.

So where's the clog in the trickle-down theory? Part of the problem begins during contract negotiations. I have this ongoing problem of thinking the other guy wants a fair deal. The sad truth is that the last time I was involved in a negotiation where both parties were interested in a fair outcome was when I traded my kid brother a water pistol for his Slinky. Ever since, I've been cheated, lied to, ripped off, folded, spindled, and mutilated. It seems like almost everybody I meet in this business is ripping off everybody else.

One common rip-off is the "bogus bonus" technique, and here's how it works. During an interview at some big game company, they tell you how important the project is, how important you are to the successful completion of the game, and how it's going to change the course of history when it's finished. But instead of paying you what you were making at your last job, they tell you they're going to be paying big, fat bonuses to all the team players when the game ships on time. After all, they say, this makes it fair to both them and you. So you agree.

illustration by Jackie Urbanovic

Halfway through the project they wail about being short on cash, they lay off half the company, move the delivery date up while adding features, and are, of course, forced to cancel bonuses. You manage to get a half-baked game out the door by working 80-hour weeks, but the big, fat bonus is long gone. The publisher got half of your work for free because you only got paid for 40 hours of those 80-hour weeks. One big company owner in L.A. has run this scam four years in a row, taking home a million-dollar bonus for himself every single year, according to the company's SEC filings.

So how does an honest developer avoid this fate? Simply decide what's the minimum cash you want after working 40 hours a week, then multiply that by two. This is the minimum salary you can settle for (because you'll be working 80 hours a week, remember?). If they offer you a bonus, ask them a few questions, such as how many people got paid bonuses last year. You don't really care what the answer is, you just want to see their reaction. If asking questions makes them uncomfortable, sound the red alert klaxon — incoming bogus bonus.

Another rip-off is the "complicated conundrum" scam. This one is run by a programmer against the producer. The programmer is running late on an important part of the project. The producer politely asks the programmer if hiring a couple more programmers would help get that part of the project back on track. The programmer then explains that bringing on new hands would mean he'd have to explain the extremely complicated code to the new people, and by the time he explained the whole thing he could have just finished it all up by himself. The producer thinks this makes sense.

As the project continues, more parts

continued on page 63.

Mike Kelleghan has a few uses for dishonest developers. Pernicious publishers he uses as filler in the cracks left by the last L.A. earthquake, malingering musicians are chopped up for kitty litter, crooked coders are ground into fertilizer for the nature preserve in the backyard, and amoral artists are hung up as window coverings to keep out the light. Honest developers (yeah, both of you) are invited to partake in ruminations about the good old days at mkelleghan@compuserve.com.

continued from page 64.

are late, until it seems like every part is late and the whole project is way off schedule. By the time the producer figures out what's going on, the project is years late, the developer got an extra year or two out of his or her contract, and the game ends up canned.

What's an honest producer to do? First, keep a log of every task you assign, and the estimate to completion you get back from the developer. Calculate how far off the actual completion date is from the developer's projected date as a percentage. This is the developer's "error rate." Second, have the star developer do about 75 percent of a given task, and then hand it off to a junior developer to clean up, debug, test, and finally submit. The objective is that no single person is responsible for the entirety of any single task. This allows the junior developer to learn from watching the senior, frees the senior to focus on what he or she does best, and generally allows honest developers to grow and expand in their specialty.

When a dastardly developer springs the "complicated conundrum," you can pull out the log of the developer's error rate. A good developer will have a very stable error rate. A poor developer, or a crooked one, will be late by different percentages every time. With this information you can say, "Gee, every time you tell me how long it's going to take, you're wrong by *x* percent, so based on these figures I can hire two new guys, have them look over your code without bothering you, and still have time left over." If your developer is a crook, this might be a good time to duck.

The dastardly developer can't threaten to quit and leave your project high and dry, because you have a couple of junior programmers that have been following in his or her footsteps and have a good handle on what's going on. Fire the turkey, hire a new guru, and when the project finishes on time, remind the publishers about that "on-time" bonus they promised you. ■

ADVERTISER INDEX