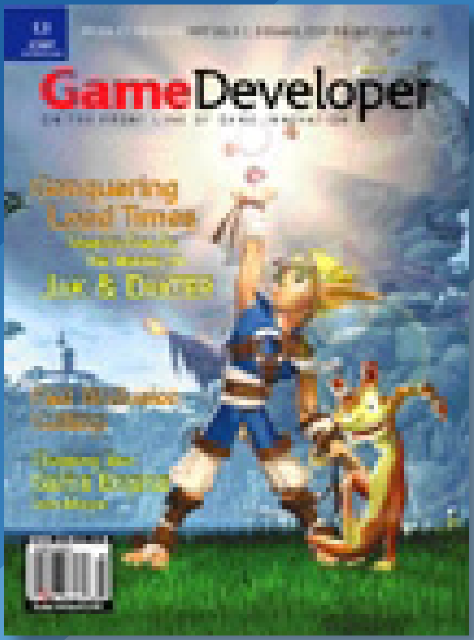
GAME DEVELOPER MAGAZINE

APRIL 2002



UBM Tech

Return

About Face

High-powered consumer 3D hardware continues to blaze forward, and I, like many, am duly impressed by the new opportunities for realism afforded by each advancement. We have solved a lot of long-standing problems obstructing believably realistic real-time environments and are tantalizingly close to conquering many others.

Clever technology notwithstanding, one of the quickest ways to dash the realism of even the most richly immersive 3D environment is to put a human in there and have him say something.

Audiences and critics alike were dazzled by the visual realism of *Final Fantasy: The Spirits Within*, but despite the heroic efforts on the animators' and technicians' parts, a number of critics pinged the film for lacking believability in the facial animation: expressions that missed their mark in some barely perceptible way, lip-synching that seemed stilted, lifelessness beaming from meticulously shaded eyeballs.

Going back to the Latin, to animate something is literally to imbue it with a soul. Humans are very sensitive to the nuance of expression they and their kind are capable of; it's a form of communication more essential to our survival than language. Darwin studied human facial expressions and believed our ability to interpret them represented the existence of a continuum between the human and animal worlds, between nature itself. It is far more difficult to quantify that kind of subtlety, so close to our very essence, than it is to create a few million vertex-shaded blades of grass.

With every technological advancement, are we getting closer to or farther away from the goal of creating truly realistic, believable virtual humans? As we approach perfection in virtual environments, do our senses then amplify perceived shortcomings of the human element? Is it a psychological barrier that our conscious minds know the person we're observing doesn't actually exist? Just where is the solution to this problem going to come from?

On the one hand, this industry has always loved a challenge; perhaps the challenge of convincingly simulating the range of human expression (dozens of intricate facial muscles creating between 5,000 and 10,000 discrete, recognizable expressions, according to estimates from psychological researchers) is one we'll be able to solve once and for all when a sufficiently high number of transistors arrives. Perhaps all our gorgeous, soulless figures need is a few million more polygons and a fresh set of shading algorithms and they'll be a bunch of virtual Brandos.

On the other hand, we can accept the notion that every medium has its strengths and shortcomings relative to other media. Traditional cel animation today retains many of the same abstract stylization conventions for facial expression and lip-synching that have existed since the birth of the medium, in part because viewers recognize them so well. In the near term, stylization affords game developers the ability to circumvent an insurmountable hyperrealism with creative flair. New hardware has done a lot to encourage visual creativity: 3D cel-shading, 2D "paper" characters such as Paper Mario or Parappa the Rapper, and plenty of research on the non-photorealistic rendering front.

Infinite realism is a tricky goal for human expression in games. Facial animation, while far better than it used to be, is still fraught with major compromises to genuine immersiveness. Engineers, animators, psychologists, anthropologists, and neurologists are all going to continue working to understand and thereby convincingly replicate every facet of human expression; there's certainly no reason to stop, even though I think the solution will continue to evade us for years. For now, it's exciting to see other creative avenues open up and flourish at a time when we are still exploring the creative and artistic potential of our young medium. Because who will ultimately judge when computers have successfully conquered reality?

Jennifer

GameDeveloper

600 Harrison Street, San Francisco, CA 94107 t: 415.947.6000 f: 415.947.6090

Publisher
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL

Editor-In-Chief
Jennifer Olsen *jolsen@cmp.com*

Managing Editor
Everard Strong *estrong@cmp.com*

Production Editor
Olga Zundel *ozundel@cmp.com*

Product Review Editor
Daniel Huebner *dan@gamasutra.com*

Art Director
Audrey Welch *awelch@cmp.com*

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

Contributing Editors
Jonathan Blow *jon@bolt-action.com*
Hayden Duvall *hayden@confounding-factor.com*
Noah Falstein *noah@theinspiracy.com*

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

ADVERTISING SALES

Director of Sales & Marketing
Greg Kerwin e: *gkerwin@cmp.com* t: 415.947.6218

National Sales Manager
Jennifer Orvik e: *jorvik@cmp.com* t: 415.947.6217

Senior Account Manager, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 415.947.6224

Account Manager, Northern California
Susan Kirby e: *skirby@cmp.com* t: 415.947.6226

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

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

Account Representative
Aaron Murawski e: *amurawski@cmp.com* t: 415.947.6227

ADVERTISING PRODUCTION

Vice President, Manufacturing Bill Amstutz
Advertising Production Coordinator Kevin Chanel
Reprints Stella Valdez t: 916.983.6971

GAMA NETWORK MARKETING

Senior MarCom Manager Jennifer McLean
Marketing Coordinator Scott Lyon
Audience Development Coordinator Jessica Shultz

CIRCULATION

BPA INTERNATIONAL
Game Developer is BPA approved

Group Circulation Director Catherine Flynn
Circulation Manager Ron Escobar
Circulation Assistant Ian Hay
Newsstand Analyst Pam Santoro

SUBSCRIPTION SERVICES

For information, order questions, and address changes
t: 800.250.2429 or 847.647.5928 f: 847.647.5972
e: *gamedeveloper@halldata.com*

INTERNATIONAL LICENSING INFORMATION

Mario Salinas
t: 650.513.4234 f: 650.513.4482 e: *msalinas@cmp.com*

CMP MEDIA MANAGEMENT

President & CEO Gary Marshall
Executive Vice President & CFO John Day
President, Technology Solutions Group Robert Faletra
President, Business Technology Group Adam K. Marder
President, Healthcare Group Vicki Masseria
President, Specialized Technologies Group Regina Starr Ridley
President, Electronics Group Steve Weitzner
Senior Vice President, Business Development Vittoria Borazio
Senior Vice President, Global Sales & Marketing Bill Howard
Senior Vice President, HR & Communications Leah Landro
Vice President & General Counsel Sandra Grayson
Vice President, Creative Technologies Philip Chapnick

CMP United Business Media | GamaNetwork

WWW.GAMANETWORK.COM

SAYS YOU

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

Understanding Is Key

I read "Mathematical Growing Pains" (Inner Product February 2002) with great interest.

Alas, I'm a 3D artist, and I can't really say that I can program (though I'm reading through C++ books and understanding it little by little) and have very little math knowledge beyond high school calculus.

More than anything, I'm rather proud that I had the faculty to sort out the meaning behind "why we should care about *n*-dimensional generality," and though I lacked the engineering background to appreciate Blow's article in full, I am a stickler for *understanding* things — not just plugging numbers into formulas or mashing buttons in a 3D application. I think it is that part of me that responded to his article the most.

There are plenty of barely qualified hacks in any and every department of every field, and I think Blow's exhortation to seek understanding in underlying concepts is relevant to everyone at all levels. If that "negativity" he mentioned in his column was indeed a reflection of general ignorance and it was toned down, I am really very glad it was not removed altogether.

Jin Choung
via e-mail

Mathematical Quibble

I recently read Jonathan Blow's Inner Product column, "Mathematical Growing Pains" and found something funny. Blow states, "As a bonus, the surface normal of the line is (a, b), and the distance from the line to the origin is c" (1st page, 2nd column, 3rd paragraph). Since I was taught exactly the same thing in school, I didn't give it much importance at first glance, but then I started thinking more thoroughly about it and realized that despite the fact that the first statement is true (mostly because is it the gradient for a $z = ax + by$ equation), the second one is false. The distance to the origin isn't the same for both $2x + 2y + 1 = 0$ and $x + y + 1 = 0$. The correct expression should be abs($c/\sqrt{a^2+b^2}$).

That really astonished me, because I have the c== distance definition memorized and never really cared about it.

Manuel Sanchez
via e-mail

I think Blow's exhortation to seek understanding in underlying concepts is relevant to everyone at all levels.

JONATHAN BLOW RESPONDS: *Well, I didn't really explain this well in the article, though I guess I should have. As you know, the equation $ax + by + c = 0$ is underconstrained, and there are an infinite number of possibilities for any particular line (they are all multiples of each other). Usually we choose the one so that (a, b) is a unit vector. I had assumed that this is the case (I say that it's the "surface normal" to the line, and in graphics, a surface normal is assumed to be unit length).*

When (a, b) is a unit vector, c is the distance to the origin (because $\sqrt{a^2+b^2}$ becomes 1, so this is consistent with what you were saying).

But you can store the line equation so that (a, b) is not unit, so my article is a little misleading there, I guess. I hope people new to this kind of math don't get too confused by that.

On Industry Awards

Re: "The Envelope, Please," Hal Barwood's Soapbox on game industry awards (February 2002).

The Academy of Interactive Arts and Sciences (AIAS) has been working for several years to meaningfully represent the interests of the game development community and to offer meaningful and prestigious awards to our professional community.

The various craft screening committees (referred to by AIAS as Peer Panels) ensure that AIAS's individual craft award nominees are actually meaningful. These nominations are selected through arduous review and vigorous debate by some of the most accomplished professionals in the business. To be nominated for an Interactive Academy Award is to win the respect and admiration of those who really know.

I have personally been involved in the craft awards screening process for four years on behalf of the audio community. For the past two years, I have overseen the process in the Music Composition and Sound Design categories.

It might be interesting to some of your readers to take a behind-the-scenes and look at how these Peer Panels operate. HUGEsound recently posted a summary of this year's experience in its HUGEnews section: www.hugesound.com/HUGEnews.htm.

Recognition for outstanding accomplishment is important. These award nominations provide additional incentives to keep us pushing ever upward and onward in our respective crafts.

Chance Thomas
HUGEsound
via e-mail

Let us know what you think: send us an e-mail to editors@gdmag.com, or write to *Game Developer*, 600 Harrison St., San Francisco, CA 94107

Return

INDUSTRY WATCH

THE BUZZ ABOUT THE GAME BIZ | *daniel huebner*

It's official: 2001 best year yet for U.S. game industry. Thanks to the PS2, Xbox, and Gamecube, U.S. sales of videogames and related hardware hit a new all-time high in 2001, reaching $9.4 billion according to market researcher NPD. The previous record, set in 1999, stood at $6.9 billion.

NPD said that the number of consoles increased by 39 percent last year, amounting to a 120 percent increase in revenue. The higher revenue came as a result of higher sales volume on consoles, as well as the higher average price of the next-generation consoles.

On the game side, the industry hit $6 billion in sales, up from $5.4 billion in 2000. As a benchmark, the U.S. box office receipts for Hollywood movies in 2001 amounted to $8.4 billion.

The top seller in 2001 for consoles was Take-Two's GRAND THEFT AUTO 3 for the PS2, which has sold more than 2 million copies since it debuted last October. THE SIMS was the best seller on the PC, posting 2.6 million units in sales since it launched.

Nintendo was the top publisher for consoles and handhelds in 2001 (three top-10 titles, all for the Game Boy Advance). Sony had seven of the top 10 console titles for the year, and EA controlled the PC market with six of the top 10 games on that list.

Mixed messages as Fargo leaves Interplay. While Titus CEO and Interplay President Herve Caen was wishing Brian Fargo good luck in his future endeavors, Interplay itself was considering filing suit against its former chairman and CEO. Interplay's board of directors, which is accusing Fargo of improperly engaging in competition with the company by soliciting Interplay employees, has tapped Herve Caen as Interplay's interim CEO. The change in leadership at the troubled publisher is just the latest move by Titus, Interplay's controlling shareholder, to establish greater control over the company. In August, Titus increased its ownership of Interplay to 51.5 percent, and in September Titus arranged to increase its hold on Interplay's board of directors to five of seven seats.

As one leading dispute at Interplay was brewing, another was finally settled.

BioWare and Interplay have settled their differences over NEVERWINTER NIGHTS.

After four months of legal disputes between BioWare and Interplay, the two companies announced that they have settled their differences over the publishing rights to BioWare's upcoming RPG, NEVERWINTER NIGHTS. The two companies squared off against one another last September, when BioWare filed suit against Interplay to stop the publisher from sublicensing the game's distribution. Later in November, BioWare announced that it was canceling its publishing agreement with Interplay. Under the terms of the settlement, NEVERWINTER NIGHTS will be published worldwide by Infogrames, but the new agreement will be subject to certain pre-existing Interplay licenses.

Take-Two halts trading. Questions about irregularities in Take-Two's accounting procedures have erupted into a firestorm of controversy. After announcing that the company would be forced to restate earnings and revenue numbers for all of its fiscal year 2000 as well as the first three quarters of 2001, Take-Two unexpectedly postponed the announcement of its fourth-quarter results. Explaining that the company would need more time to put its accounts in order, Take-Two filed for a 15-day extension on filing its annual report.

Citing a lack of proper information regarding the company's financial reporting, the Nasdaq exchange halted trading of Take-Two shares shortly after the company postponed its fourth-quarter report. The trading hold was to remain in effect until Take-Two was able to fully satisfy Nasdaq's request for additional financial information.

Take-Two financial woes have culminated in nearly a dozen class action lawsuits filed by investors asserting that the company's accounting practices misled shareholders by misrepresenting the true state of Take-Two finances. All of the suits were filed in the United States District Court for the Southern District of New York, and charge that Take-Two issued a series of materially false and misleading statements to the market between February 24 and December 17, the day Take-Two announced that it would restate its numbers.

Get the can of bug spray, MISTER MOSQUITO is coming to town!

Eidos boutique label brings Japanese games west. The obscure world of Japan-only games will no longer be strictly the domain of import buyers. Eidos is establishing a new game label dedicated to bringing Japanese-only videogames to the rest of the world. The Fresh Games label will launch with three Sony-developed Japanese titles previously unavailable outside of Asia: MISTER MOSQUITO, MAD MAESTRO!, and LEGAIA 2: DUEL SAGA. Fresh Games titles are set to debut in spring 2002.

UPCOMING EVENTS
CALENDAR

ELECTRONIC ENTERTAINMENT EXPO

LOS ANGELES CONVENTION CENTER
Los Angeles, Calif.
Conference: May 21–23, 2002
Expo: May 22–24, 2002
Cost: $200–$475
www.e3expo.com

Expo admission free to game industry professionals

Virtools Dev 2.0

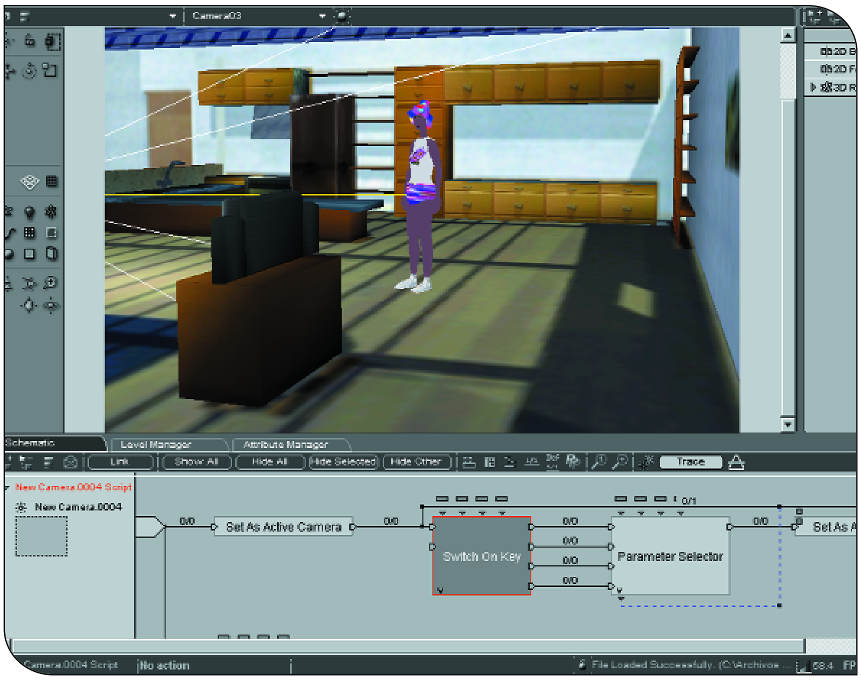
by daniel sánchez-crespo

The second version of Virtools clearly builds upon previous releases, keeping a similar look while adding a host of new features that greatly improve the final package. For those new to the product, Virtools is a production environment (such as Director) designed to allow the rapid creation of 3D-interactive applications.

It features importers for many popular file formats, an intuitive drag-and-drop scene manager, and a behavior engine that allows users to assign behaviors graphically to the different entities in the world. Import your Character Studio files (other supported formats are discussed later), make them move by dragging the appropriate behavior from the palette, and off you go. No recompiling, no coding: all the actions are performed graphically.

For this reason, Virtools is a great tool for game designers needing to create working prototypes. In fact, it works so well that it can be used to create full-blown games of moderate size (no, that does not include EVERQUEST). Thanks to the web player supplied with the package, Virtools can create some of the best web-based content around, with top-notch 3D graphics and interaction. It's rather impressive to see some of the Virtools-based DOOM clones available online.

Version 2.0 offers a wealth of new stuff for Virtools aficionados. To begin with, the number of behavior building blocks (BBs) has expanded to a whopping 400. That includes everything from basic transforms or texture assigns, to rich and complex behaviors such as skin/bones controllers and Bézier patch support. Another highlight of the new features is the way Virtools handles progressive meshes. You can adjust your triangle count by moving one slider and modifying it on the fly to guarantee a good frame rate. Another impressive feature is the portal system that culls away large parts of the geometry to optimize rendering speed. All these features give Virtools the ability to produce professional-looking applications.



VIRTOOLS DEV 2.0. Tracing through the building blocks (active ones are shown in red) allows users to debug applications done in Virtools.

Should you require additional behaviors, there are several tracks you can follow. Your first stop should be to investigate purchasing the add-on Physics Pack (new in version 2 of Virtools), a version of the Havok toolkit designed to run with Virtools. The Physics Pack adds 30 new BBs, which implement gravity, friction, and the like in your project. If you don't want to spend the extra cash to get more behaviors, Virtools sports a large fan base, and many community web sites have free downloadable behavior and composition collections. You can also create new behaviors using the supplied SDK.

As far as the interface goes, the overall philosophy has remained unchanged since

DANIEL SÁNCHEZ-CRESPO | *Dani has been in involved with computer graphics in one form or another for about 10 years, from raytracing to academic research and now as a professor of game programming. He is founder and lead engine programmer at Novarama, an indie studio in sunny Barcelona, Spain. You can reach him at dani@novarama.com.*

✶✶✶✶✶ excellent
✶✶✶✶ very good
✶✶✶ average
✶✶ disappointing
✶ don't bother

the previous release: three stacked panes display the scene, behavior library, and current script. All actions are performed via drag-and-drop, with no coding required. Still, the interface has several problems that have remained from previous versions. For one thing, it does not follow a "standard" Windows design. In addition, the interface's structure is counterintuitive: it lacks an undo feature, it does not follow standard key mappings, and there is a huge amount of information displayed on-screen. All these issues can make for a long and steep learning curve. An extra effort could have been made to make this otherwise strong product a bit more accessible.

Workflow with Virtools is, despite the flaws of the interface, a breeze. After importing your content (.3DS, .X, .MP3, .WMA, and Maya files are among the formats supported) you can begin assigning scripts to the items by dragging building blocks to the lower pane. Click the play button, and your composition will run interactively within the environment. You can even select the renderer of your choice (DirectX 5, DirectX 7, or software/hardware OpenGL; DirectX 8.1 support is being added with the imminent 2.1 release) and limit your frame rate should you need to do so. While you are in play mode, the Trace feature shows you which BBs are activated at any given time, their parameter values, and other information. You may also set breakpoints, which will pause the execution once it reaches the marked BB. This way you can avoid losing vital information about its inner workings when the script is running at full speed. All these debugging features make working with Virtools very easy.

The documentation is a mixed bag. Virtools comes with a printed manual that, aside from being slick and well laid out, does a great job of introducing us to the basic components and terminology. The 230-page manual covers the program operation and interface as well as the inner structure of the behavior engine in a fair amount of detail. It also features two complete tutorials that guide users through the different features of the system. Despite offering a strong introduction, however, the manual is not sufficient to gain real knowledge of the system: I had to depend on online help files, context-sensitive help, and the like to learn about specific building blocks and — surprisingly — the SDK.

One of the main strengths of Virtools Dev is the access to the SDK, allowing users to create specific building blocks. Do you want to create a complex AI? Then you need to use the SDK. Sadly, the SDK is not even mentioned in the manual, and the available online documentation is "under construction." Several extensive samples are supplied (an AVI player, a Max exporter, among others), but if you want to go on with the SDK, you are left on your own.

The Bottom Line

Virtools 2.0 is a great tool with some annoying problems. On the plus side, it allows you to create great interactive content rapidly with its intuitive building block system. Being a programmer, I'm more used to coding, but it's good to know that there's something out there worth using for those of you that don't know (or like) down-and-dirty coding. The workflow is very intuitive and, given some time, can produce stunning results. In fact Microsoft selected Virtools as the first official Prototyping Tool: you can have a working prototype with final art and gameplay within weeks, not months, and if the game is good enough for Xbox, you can then move on to a final working environment (or stay with Virtools should you wish to do so), while keeping all of the art assets you have created. This preproduction or production-planning method, greatly encouraged by Virtools, can be a real advancement from traditional production techniques.

On the minus side, the first hours with the system are rather harsh: the interface is not intuitive, and the manuals only cover part of the picture. It's true that using the manual tutorials can get you up and running fast, but Virtools needs more and better documentation. I have the sense that many interesting features are hidden in there somewhere, but the lack of documentation is preventing me from discovering them. And that's a shame.

EQUILIBRIUM'S DEBABELIZER PRO 5.0

by tom carroll

Reviewing DeBabelizer Pro 5 for game development in the amount of space allotted here is a challenge. The product as a whole is as deep as True Color and as broad as a panoramic JPEG. But it may not be right for every game developer, especially one currently using DeBabelizer 4.5.

VIRTOOLS 2.0 ✶✶✶

STATS

VIRTOOLS S.A.
Paris, France
+33 (1) 42 71 46 86
www.virtools.com

PRICE

Virtools Dev 2.0: $5,000
Physics Pack: $5,000

SYSTEM REQUIREMENTS

Pentium II or equivalent with 64MB RAM running Windows 95/98/ME/2000/NT 4 (SP 6)/XP; CD-ROM drive, monitor capable of displaying 1024×768 in 16-bit color (65,536-color/high-color), pointing device, Direct3D- or OpenGL-compatible graphics card with 8MB RAM; DirectX 5.0 required for DirectX-compatible graphics cards, Internet Explorer 4.0 required for the online reference.

PROS

1. Simple workflow.
2. Robust feature set.
3. Good entry point for Xbox.

CONS

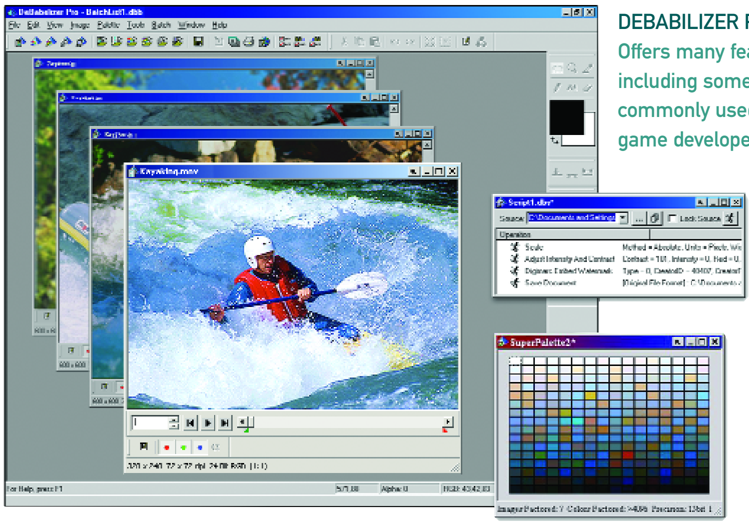
1. Counterintuitive, nonstandard interface.
2. Insufficient documentation for everyday use.
3. SDK documentation is poor: unfinished reference help file, no tutorials.

DeBabelizer Pro was originally developed to automate the tedious tasks of graphics production: image processing, graphics optimization, and file conversion. The package combines simple scripting of repetitive tasks, batch processing of scripted processes, and advanced automation into one seamless whole. All this power comes at the expense of your neat little desktop. The interface can get cluttered up rather quickly, especially when multiple images are open simultaneously. Oh, and the toolbar isn't terribly intuitive. But once you get past these shortcomings, the package itself is wonderfully deep.

For game developers, especially those working on console systems that still require texture optimization (anyone out there working on PS2 games, raise a hand), DeBabelizer is the answer. It's a must for anyone porting graphics from PC to console, optimizing palettes, performing blue- or green-screen compositing, eliminating scan lines, and/or converting assets from NTSC to PAL or vice versa. The package supports more than a 100 file formats for input and output. Along with fairly pedestrian .GIF, .JPG, and .PNG formats, you can also manipulate more exotic Ventura Publisher .IMG and Dr. Halo .CUT formats, among others. While game developers don't necessarily need so much flexibility, it's comforting to know that collecting textures from exotic sources isn't going to become a time pit.

The package has also grown famous for pioneering such features as batch processing and SuperPalette creation.

But no software package exists in a vacuum. Following DeBabelizer's lead, other packages, such as Photoshop, now provide similar functionality for roughly the same price. (DeBabelizer Pro 5 is available for Windows XP/2000/NT/Me/98 as well as Mac OS 9.x and X at a suggested retail price of $699 and a street price of $479.) And that brings us to the current state of DeBabelizer affairs: DeBabelizer Pro 5.



DEBABILIZER PRO 5 Offers many features, including some not commonly used by game developers.

DeBabelizer Pro 5 does everything past versions have done, plus enabling users to do some new things, such as automatically digitally watermarking high volumes of images and multi-frame files on the fly.

Pro 5 also features the ability to publish web catalogs with HTML, thumbnails, and JPEG files; new Palm OS and Windows CE color palette optimization; QuickTime, AVI, and RealVideo file conversion and compression; and increased batch automation. While many of these new features — especially web cataloging and watermarking — are powerful, they are aimed primarily at markets other than game development.

The real question is, then, could a game developer already using Debabelizer 4.5 get by without the new package and the $149 upgrade fee? Probably. Version 4.5 contains the core functionality that most game developers require.

Should a game developer who is not using DeBabelizer consider trying to make Photoshop (or some other equivalent) do the same job? Possibly. But would it be better for that person to invest in DeBabelizer Pro 5 because it is the best in the business? Absolutely.

As a final note, I considered giving DeBabelizer Pro 5 just three stars, but decided it's unfair to penalize a software developer for attempting to broaden its market by adding new features, regardless of how little they apply to game development.

✶✶✶✶ | DeBabelizer Pro 5
Equilibrium Technologies
www.equilibrium.com

Tom Carroll is a 2D/3D artist who would be quite happy if he could somehow fit 25 (or more) hours in each day. Reach him at jetzep@pacbell.net.

MAXON COMPUTER'S BODY PAINT 3D

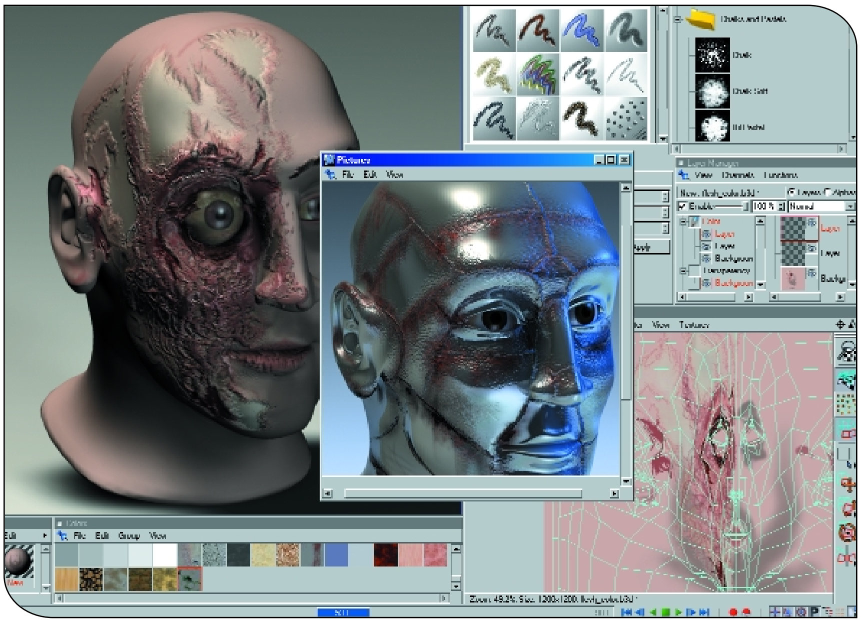
by steve theodore

German developer Maxon Computer's Body Paint 3D is the newest entry into the field of 3D texture painting. The $595 program (Windows and Mac OS) allows users to paint complex, multi-layered texture maps directly onto 3D objects.

Body Paint is designed to work with models created in other packages, although it can be integrated into Maxon's flagship 3D software, Cinema4D XL. Maxon offers free plug-ins for Lightwave (versions 6.5 and 7) and 3DS Max (3 and 4), allowing direct import and export of models and materials. Models can also be imported and exported via standard 3D interchange files, including .DXF, .OBJ, and .3DS, or more esoteric formats such as VRML and QuickDraw 3D. Textures can be imported and exported as Photoshop .PSD, .TIF, .TGA, .JPG, or Macintosh-standard PICT files.

An arbitrary number of views of the model can be displayed and painted simultaneously. While navigation seems sluggish compared to 3DS Max or Maya, performance is adequate for the limited task of finding views to paint. Extensive

✶✶✶✶✶	excellent
✶✶✶✶	very good
✶✶✶	average
✶✶	disappointing
✶	don't bother


BODY PAINT 3D's interface handles a great deal of information with strong workspace-management tools.

hotkey and customization support, plus good window- and palette-management tools, make workspace management fairly easy, although the reversed mouse-pitch for camera movement is off-putting (a right mouse move rotates the view right, not left, as users might be accustomed to in their 3D modeling package).

The workflow is quite similar to traditional 2D painting; the paint tools are responsive, and the feedback is very fast on a modern machine. The program offers a large collection of brushes, but all of them (airbrushes, charcoals, pencils, and so on) are customized instances of a single generic brush, similar in capability to the default brush in Painter. The interface for customizing and storing brushes is spartan but functional. Users can also define texture brushes (the familiar Rubber Stamp or Pattern tools from Photoshop) and the very handy "multibrushes," which affect multiple texture properties (such as color,

✱✱✱✱✱	excellent
✱✱✱✱	very good
✱✱✱	average
✱✱	disappointing
✱	don't bother

bump, and specularity) at the same time. Other standard paint tools — selection marquees, friskets, magic wands and so on — all function as expected. Each texture can include an arbitrary collection of Photoshop-like layers with masks and multiple blend modes. A selection of 2D image filters (color correction, sharpen, and so on) rounds out the paint functionality.

Successful 3D painting depends on the model's UV mapping; poorly arranged UV coordinates can result in smeared textures or seams between adjacent polygons. Body Paint includes tools for touching up or completely replacing UV mapping on a model. One standout feature is Optimal Mapping, which distributes texel density evenly across an object to prevent texture smearing. This provides nice results but requires hand-tweaking to avoid prodigal waste of texture memory. To accommodate on-the-fly edits of existing maps, Body Paint can reposition existing textures to match edited UV coordinates. While this process can result in seams and filtering artifacts, the results are generally as good as could be had from performing the same edit by hand. Overall, the UV tools lag behind those in Maya or Texture Weapons in sophistication, but being directly integrated into the 3D paint application makes them extremely useful nonetheless.

Unlike similar tools, Body Paint 3D also includes a full-featured rendering system. The Raybrush renderer offers a fast raytracer and cel-shader capabilities. Users can also render into a 3D paint window and work directly on the rendered image, with the high-quality filtering and refractive effects of the raytraced image. Real-time artists will find, however, that the power of the renderer is irrelevant to their immediate needs.

Overall, Body Paint 3D is a capable program at an attractive price. The painting toolset has few frills but is capable, and the UV tools are solid. Painting performance is fine, and the interface is stylish. The most significant competitor, Right Hemisphere's Deep Paint 3D, does hold two key advantages: more sophisticated UV-editing tools and a projective painting mode that does an excellent job of eliminating texture seams (see Product Reviews, August 2001). Nevertheless, Body Paint does a workmanlike job for half the price of the Deep Paint 3D/Texture Weapons bundle. Teams on a budget, teams using Macintoshes, and current Cinema 4D users should all look seriously at Body Paint 3D.

✱✱✱½ | Body Paint 3D
Maxon Computer
www.maxoncomputer.com

Steve Theodore is an animator and character designer at Valve Software. He can be reached at stevet@valvesoftware.com.

Return

PROFILES

TALKING TO PEOPLE WHO MAKE A DIFFERENCE | jennifer olsen

Doug Church: Rummaging Through the Designer's Toolbox

If you don't know who Doug Church is, don't blame us. Doug has done design and programming work on three of *PC Gamer*'s top 20 PC games of all time: ULTIMA UNDERWORLD (I and II), SYSTEM SHOCK, and THIEF: THE DARK PROJECT. In his 10-year career he has frequently found ways to allow cutting-edge technology to produce innovative gameplay experiences, mitigating the long-running debate that one can only come at the expense of the other.

Doug Church is lobbying for a common vocabulary for game designers.

GD. You've been doing work for several years on formal abstract design tools. Are game designers getting any closer to benefiting from a common language?

DC. I think what has happened has been a lot of individual efforts from people to move it forward, but no common front. Harvey Smith on DEUS EX 2 has tried to create some common abstract vocabulary for that team, for instance. This year at GDC, Bernd Kreimeier is running a roundtable on patterns for design. One of my hopes is to have a plan for really moving this forward at GDC next year, hopefully taking some of this year's discussion and using it as a basis for several talks and roundtables next year.

GD. You've been doing some work with the academic community through IGDA. What are you learning about academia's needs from the game development community?

DC. A big one is a curriculum framework for academics wanting to build development-related coursework.

GD. Which area of game development curriculum development poses the greatest challenges?

DC. The issues of trying to introduce any new curriculum or courses into a modern university poses the largest challenges. But if we ignore that logistical and political problem, I'd say the multi-disciplinary nature of game design is hard to make work in a university context — not that some people aren't trying hard and making good progress.

This is another area where better vocabulary and analysis would help. Right now teaching game technology is fairly straightforward, as there is a body of well-understood knowledge. The design space is far less clear, far less agreed upon, and far less concrete. This makes teaching it hard.

For now, this means most university design work is very piecemeal. And often the valuable part of developer interaction with academia is simply explaining how the industry really works, and what the parameters and possibilities are.

GD. How are the changing content demands for the current generation of consoles and PC hardware changing the task of game design?

DC. As our capabilities grow, content creation has become more time-consuming and specialized. It is easy to generate vast fields of tedium with ever higher poly counts. Normal maps and so on are a scheduling and cost headache, but not hard. Creating environments rich in meaningful user interactions is hard.

Sadly, progress doesn't happen in all game aspects at once. Fifteen years ago, a character was a 2D bitmap, and emotions were indicated through text messages. Now, that character is an articulated 3D figure, and we are just beginning to reach the point where facial expressions can express things like crying. And our ability to generate and simulate reasons for emotions is still pathetic.

Representation and feedback are crucial aspects of good game design, and the old standards of text messages and die rolls don't cut it anymore. As technologies advance, those that haven't moved forward are exposed. You can't just say, "You critically missed," or "Wounded 42%," you need to figure out how to show the player what happened, and why, all in the context of the world.

This is where tools for design become more vital. Building a space appropriate for a given game style is a mix of good architecture, well-tuned game systems, and steady information flow to the player. This requires the flexibility to experiment with and hone design ideas.

GD. Doesn't that end up venturing into design-on-the-fly territory?

DC. Well, there is a difference between flexible design with goals in mind and design-on-the-fly. That is one of the reasons we need to develop some way of speaking about design, so we can speak about design goals, and what tools we have to change the players' experience.

It is almost impossible to write a design doc for any brand-new project that is both accurate and specific. Design docs have specifics, which are hopes, and they have general goals. The specifics may change as the project goes on, as you learn more about what actually works to achieve your goals.

If the tech specs change, you may have to rework many of your specific design decisions. And that is why tool flexibility is paramount. You must have tools good enough to change your specifics routinely, while keeping the overall goals in mind.

There is a reason good games often went through extensive play-test and tuning. Good design usually means an understanding of where you want to go and a willingness to change and experiment until you get there. But that change can't reasonably be hand-editing some hex file anymore.

Return

Inverse Kinematics with Joint Limits

This month and next, I'm going to talk about inverse kinematics (IK). Jeff Lander wrote a series of introductory articles about IK in 1998; see For More Information at the end of the article if you'd like to get caught up.

There are a few different paradigms for solving IK problems, but I'll concentrate on Cyclic Coordinate Descent (CCD). CCD is easy to implement and is fairly modular; in other words, you can hack stuff into it without too much trouble.

How CCD Works

CCD is an iterative numerical algorithm. You have a chain of joints, which I will call the "arm"; the arm is in some initial state, and you want the end of the chain (the "hand") to reach some target. We have some measure of the error in the current state of the arm, which involves the position and orientation of the hand with respect to the target. CCD iterates over the joints in the arm, adjusting each in isolation, with the goal of minimizing this overall error (see Figure 1).

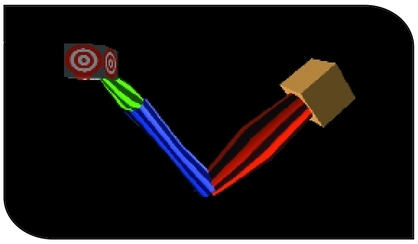


FIGURE 1. A CCD solution for an arm, anchored at the mustard-colored block, to reach the designated target. Green: hand; purple: forearm; red: upper arm.

To animate a human character, we want CCD to give us solutions that are valid for a human body. When picking a glass up from a table, we don't want the character's elbow to bend backward in a physically impossible way. In order to prevent this situation, we can enforce limits on the ways each joint can bend. A shoulder has all three degrees of rotational freedom, but each degree is limited — the shoulder can only reach through a certain range of angles and can only twist so far. An elbow might be modeled with two degrees of freedom: one that bends the forearm toward the upper arm, and another that twists the forearm.

CCD Convergence Issues

Because CCD is iterative, you can enforce these limits at each step by taking the orientation for any joint and forcing it to stay within the valid range. However, this affects CCD's ability to converge on an answer. CCD is a blind hill-climbing algorithm; it is walking across a terrain defined by your error function, trying to find the lowest point. But when you impose joint limits, you stick invisible walls on the terrain of which the hill-climber isn't aware. A typical situation occurs when the system tries to reach the target by expanding the elbow beyond its joint limit. After each iteration the elbow is forced back within limits, so the algorithm doesn't get anywhere.

We can detect this situation and attempt to resolve it by starting the arm in a different configuration in which that particular wall will no longer be in the way. This is a special case of a time-honored tool called simulated annealing.

Using the most basic form of simulated annealing, you would reset each joint to a random pose and try again. This is probably the easiest to code. A solution that may perform better at run time is to precompute a fixed number of starting positions for the arm. During a preprocess, we can randomly choose a large number of points in space; for each target point, we discover a starting configuration for which IK is sure to converge. For experiments I've tried with reasonable arms, four starting positions is enough. You cluster the group of test points reachable from each starting state. At run time, given a target in arm-relative space, you find the cluster centroid that is closest to the target and use the corresponding initial state.

In this month's sample code (available at www.gdmag.com), though, I just chose the starting points by visual inspection. I interactively moved the target around until I found a situation that caused the solver to fail. Then I would find a nearby target for which convergence was successful and use that successful final arm state as the initial state for the new target. This lackadaisical approach may actually be fine for most projects, because not many starting configurations are needed. (It is important to realize here that I am restarting the arm in a neutral position every frame. If

JONATHAN BLOW | *Jon is a game technology consultant living in San Francisco. He reads e-mail sent to jon@bolt-action.com. Game that influenced this article: Jumpy McJump, by gameLab!*

you choose to begin the CCD solve from an unpredictable pose, this problem becomes more difficult.)

Implementing Joint Limits

How do we allow the animator to express joint limits, and how do we write the code to enact those limits? We know that commercial animation packages such as Maya and 3DS Max will do IK; we might follow their examples. But those systems don't have very intuitive interfaces, and they don't provide good control over what the IK solver does. Let's roll our own.

Nicer Rotation Decomposition

Common IK packages allow you to express joint limits by clamping Euler angles, componentwise, into adjustable intervals. But the resulting set of valid directions for the bone is kind of weird, and you have minimal control over its shape. Consequently, the IK solver comes up with solutions that you don't want, and you need to fix them up. This process is tedious even in offline animation; it's unacceptable for an interactive game. So I'm going to discuss an alternative rotation decomposition and a means for limiting rotations that is cleaner than that of Euler angles.

As I discussed in my February 2002 column ("Mathematical Growing Pains"), any two (noncolinear) unit vectors a and b define a plane. We can construct a simple rotation that rotates a onto b but does not affect vectors orthogonal to the plane. The quaternion form of this rotation is the square root of the Clifford product ab; you can compute the quaternion with a dot product, a cross product, and a little bit of trigonometry (see David Hestenes' book in For More Information). If we wish to limit a bone rotation R, we can factor it into two rotations: one is the simple rotation that moves the bone into its final direction vector, and one represents the twist around that final vector.

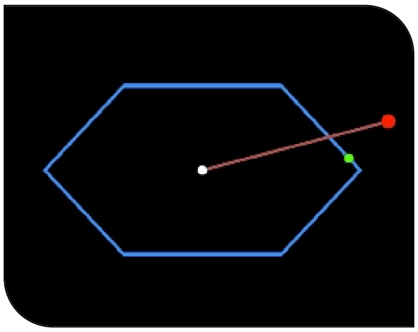


FIGURE 2. **Computing the best reach orientation as a point-in-polygon problem. The blue hexagon is our reach window. The white point is our initial state, and the red point is our goal outside the window. The valid point closest to the destination is marked in green; it is not the intersection of the hexagon with the direct path of rotation (tan line).**

We will adopt the convention that a bone is oriented along the x-axis of its local transform. We perform a CCD step that yields a rotation R for some bone, and we want to joint-limit R. Let $x' = Rx$, the direction vector in which the bone points after rotation. We first compute $S = \text{Simple}(x, x')$, the simple rotation that points the bone in the same direction R does. Now let $R = TS$, where T is a rotation representing the discrepancy between S and R. $Rx = x'$ implies that $TSx = x'$, but since $Sx = x'$, T must leave the vector x' unchanged. That is, x' is an eigenvector of T, so T is a rotation around x'. Beginner's linear algebra says that $T = RS^{-1}$.

Suppose we want to implement a shoulder joint, which has a limited amount of twist and a limited space into which the vector can reach. We compute T and S, then limit T and S separately. In this month's sample code, all rotations are represented as quaternions. To limit twist, it's a simple matter to decompose the quaternion representation of T into an axis and an angle (though we actually knew the axis already — it's x'), clamp the angle, and turn that back into a quaternion.

This decomposition is nice because it talks about rotations in terms of two concrete things that we can visualize, that are close to the things we care about for doing human body animations, and that have no hidden gotchas (or at least none nearly so bad as the Euler angle confusions). The things we think about are, "What direction does the bone point in?" and "How twisted is it?"

The Euler angle representations used by animation packages will create twist even when they don't intend to rotate around the axis of the bone. As the angles get larger, more unintentional twist is imparted toward the extremes. Our decomposition does not have this problem.

Limiting Reach

Think of the bone we're limiting as a vector pointing out into space from the origin. We can limit the "reach window" of the bone by defining some wire loop hung in space and declaring that the bone must always pass through that loop. The loop is embedded in a plane some distance from the origin. We can choose an arbitrary polygonal shape for the loop; figuring out whether the bone is inside it becomes a 2D point-in-polygon problem (Figure 2). That kind of problem is considered pretty easy to solve these days. Thus we have a versatile and easily visualized method of restricting where the bone can go.

To keep the implementation simple, I am only supporting convex reach windows; I don't see the benefit of a more complicated system at the present time. If we want our joints to be able to reach beyond a single hemisphere, we can use multiple convex windows embedded in different planes. For an alternative formulation of reach windows that supports star polygons, see the *Journal of Graphics Tools* paper by Wilhelms and Van Gelder in For More Information. They also tend to discuss a 3D "reach cone" rather than a reach window, but the ideas are equivalent. I prefer to think about this problem in 2D because it's easier to sketch on paper that way (and for other reasons that I'll demonstrate next month).

When we bang up against a reach limit, we find which side of the reach window we slammed against and then find the closest point along the segment to the des-

LISTING 1. Quickly find the simple rotation mapping *a* to *b*.

```
Quaternion fast_simple_rotation(const Vector3 &a, const Vector3 &b) {
    Vector3 axis = cross_product(a, b);
    float dot = dot_product(a, b);
    if (dot < -1.0f + DOT_EPSILON) return Quaternion(0, 1, 0, 0);

    Quaternion result(axis.x * 0.5f, axis.y * 0.5f, axis.z * 0.5f,
                      (dot + 1.0f) * 0.5f);
    fast_normalize(&result);

    return result;
}
```

tination; that point represents our final direction. Wilhelms and Van Gelder use the point where the bone slams into the wall, but it seems to me that if you do this, numerical algorithms like CCD will have difficulty finding the best point in the window. They would grind against the wall, moving very slowly.

Tuning for Speed

Converting a rotation to the reach/twist representation is computationally cheaper than using Euler angles, but it still seems to involve some expensive operations: one acos and one sin. But it turns out that we only need to perform this conversion if our rotation is trying to exceed the limit. We can develop quick and cheap tests to determine whether the rotation is within bounds; if it is, we just accept it and move on.

When testing reach, we just compute $x' = Rx$ without doing the *ST* decomposition. If x' is within the reach window, we can exit early. Checking reach still involves a point-in-polygon test, but for each reach window we can precompute the radius of a large inscribed circle, and a vector that points to the center of that circle. Then a simple dot product provides an early-accept; we only need the point-in-polygon test when x' does not pass through that circle.

This test can be even faster than we expect. *Rx* is much cheaper to compute than a normal quaternion-vector multiply (because two components of *x* are 0); but if the inscribed circle is centered around the *x*-axis, then in the end we are computing $(x \cdot Rx)$ — in other words, only the *x* coordinate of *Rx*. This requires only four multiplies and three adds: it is $(q_w^2 + q_x^2 - q_y^2 - q_z^2)$, where our rotation is the quaternion (q_x, q_y, q_z, q_w). If this quantity is greater than some precomputed constant, we know the joint's reach is safe.

We can use this same trick to accelerate the computation of the simple rotation between *x* and *Rx* (we can simplify the cross product as well as the dot product). The trigonometry involved in taking the square root of the resulting quaternion is implicitly slow, so this optimization might seem ineffective. But the square root of some quaternion *q* is the point on the 4D unit sphere halfway between *q* and (0, 0, 0, 1). In other words, it's Slerp(1, *q*, 0.5). We can use our superfast quasi-slerp from last month's column ("Hacking Quaternions," March 2002) to compute the square root of *q*.

Recall that 0.5 is a point of symmetry for linear interpolation — aside from the endpoints, this is the one spot where a lerp gives you precisely the same direction as a slerp. So we can discard much of our quasi-slerp; all we need to retain is the fast normalizer. We need to modify that normalizer, though: it needs to work for lerp distortion all the way up through 180 degrees, because this is one interesting case where we are not supposed to force our quaternions into the same half-sphere before interpolating.

I present to you Listing 1, the fastest simple-rotation-finder in the West. If the two inputs to this function are *x* and *Rx*, you can further collapse the rotation, dot product, and cross product and use the CPU savings to rule the world. (A note: The quaternion for any simple rotation starting from *x* always has an *x* component of 0). It is worth reiterating that Listing 1 computes exactly the right answer, to within the scale factor introduced by the normalization approximation.

Fast Twist Limit Testing

I've discussed how to quickly test for reach limits, but we can speed up twist limit testing as well, by not computing all of *T*. All we really need is the cosine of the twist angle, which we can test against a precomputed constant. To get this, compute $Ry \cdot Sy$. We've seen in the preceding paragraph that s is fast to compute; *Ry* and *Ry* are again cheaper than normal quaternion-vector multiplies.

FOR MORE INFORMATION

Hestenes, David, and Garret Sobczyk. *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics.* Kluwer Academic Publishers, 1984.

Wang, Li-Chun Tommy, and Chih Cheng Chen. "A Combined Optimization Method for Solving the Inverse Kinematics Problem of Mechanical Manipulators." *IEEE Transactions on Robotics and Automation* Vol. 7, No. 4, August 1991.

Wilhelms, Jane, and Allen Van Gelder. "Fast and Easy Reach-Cone Joint Limits." *Journal of Graphics Tools* Vol. 6, No 2.

Lander, Jeff. Game Developer Companion Source Page
www.darwin3d.com/gdm1998.htm

ACKNOWLEDGEMENTS

Thanks to Casey Muratori for pointing out Wilhelms and Van Gelder's Journal of Graphics Tools paper.

Strange New Worlds

Of all the worlds we are asked to create when making a game, perhaps that of the alien is the most difficult to portray successfully. On the one hand, building a world that feels truly alien requires a certain amount of unhinged creativity if we are to produce something exciting and original. On the other hand, designing an environment that really works relies on an appreciation of what makes a place appear alien while continuing to appear to be a credible setting, as opposed to looking like a cheap set from a low-budget 1970s sci-fi movie.

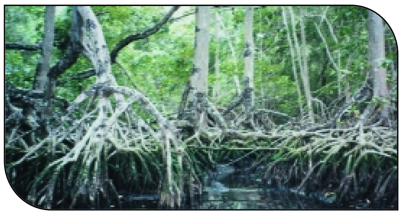
At this point, let's reach for a dictionary and have a quick look at how those distinguished men in beards (they always have beards) and women in tweed jackets (you can guarantee they're wearing tweed jackets), define "alien."

Alien: *n.* an exotic, strange being, from or characteristic of another place or part of the world; *adj.* belonging to, characteristic of, or constituting another and very different place, society, or person.

This somewhat stuffy definition of "alienness" can be put alongside the huge array of images that most of us will have accumulated from decades of fantasy art, comic books, cripplingly cheap TV shows and big-budget movies that have formalized our concepts of the alien. Using these ideas and the work of artists and designers over the years, we can look at ways of successfully creating alien worlds and how best to use our resources to achieve maximum visual impact.

There's Something Familiar About That Alien

One vital ingredient when creating something new that is designed to be interpreted visually, where it is assumed that the audience will not necessarily know in advance exactly what you're trying to achieve, is that of familiarity. In the context of this article, it may seem out of place to be talking about familiarity as an ingredient in creating something alien, but it is only through the contrast between what is recognizable and what is strange that we can make these differences work effectively.



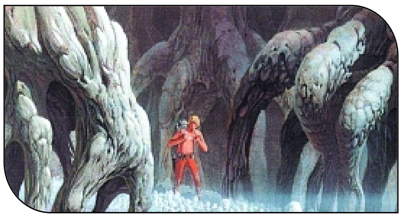


FIGURE 1 (top). Mangrove swamps in South America. FIGURE 2 (bottom). The planet Degobah in *Star Wars*.

Consider Stanley Kubrick's classic monkey-fest, *2001: A Space Odyssey*. As a child, I remember sitting restlessly through what felt like 20 hours or so of unintelligible imagery, punctuated by men dressed as apes throwing bones around, and a computer that sang "Daisy, Daisy." Watching it again as an adult, I had a much-improved understanding of the film's intent and held on firmly right up until the end. Then came the colors. I think you know what I'm talking about: Mr. Kubrick's 15-minute descent into nose-bleed-inducing psychedelia that had many children of the 1960s thinking that all the acid they'd dropped a few years before had finally caught up with them in a big, wide-screen kind of way.

This sequence, depending on whose account you read, was attempting to symbolize a journey through time, space, dimensions, consciousness, or whatever, but because it had no shred of familiarity, nothing to which the viewer could anchor some attempt at interpretation, became confusing and frustrating.

Now I know many of you will be shaking your heads, appalled at the shallowness of my appraisal of such an important piece of filmmaking. But without things degenerating into a fistfight, the point I am making is that interpretation is an important element in any visual form, but much more so in a game. A player is not a mere spectator, invited to marvel at how original and innovative our designs are. A player is actively engaged in navigating our world, and even if that world is created to be alien, it has to be understood.

One reason why the *Star Wars* universe sits so comfortably on the mantelpiece of modern movie genius is its conscious effort to use what we know about our world when creating its own. Looking at many of the environments seen in the *Star Wars* films, it is interesting to see how many of them find their origin in the vast diversity of habitats we see here on earth. The planet Degobah is a caricature of the mangrove swamps of South America (Figures 1 and 2), and the ice world of Hoth is an exaggeration of our own polar regions. Tatooine is presented as a barren world of harsh deserts, and Endor is a lush forest moon of gigantic trees. All of these places feature something we can relate to as real and takes it in a direction that distinguishes it from what we are

HAYDEN DUVALL I *Hayden started work in 1987, creating airbrushed artwork for the games industry. Over the next eight years, Hayden continued as a freelance artist and lectured in psychology at Perth College in Scotland. Hayden now lives in Bristol, England, with his wife, Leah, and their four children, where he is lead artist at Confounding Factor.*

used to. Artists often achieve this effect through exaggeration or by adding things that are unfamiliar to us.

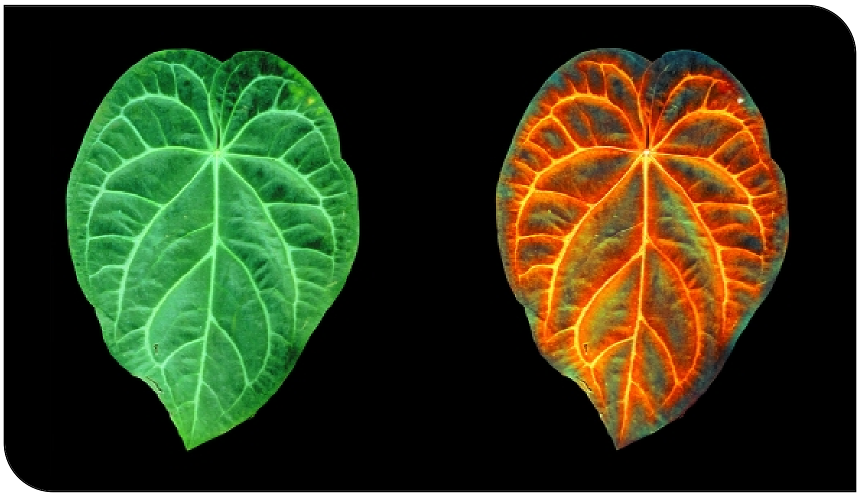
This process is certainly not unique to Mr. Lucas's lucrative franchise, and its principles can be very useful when designing an environment for a game.

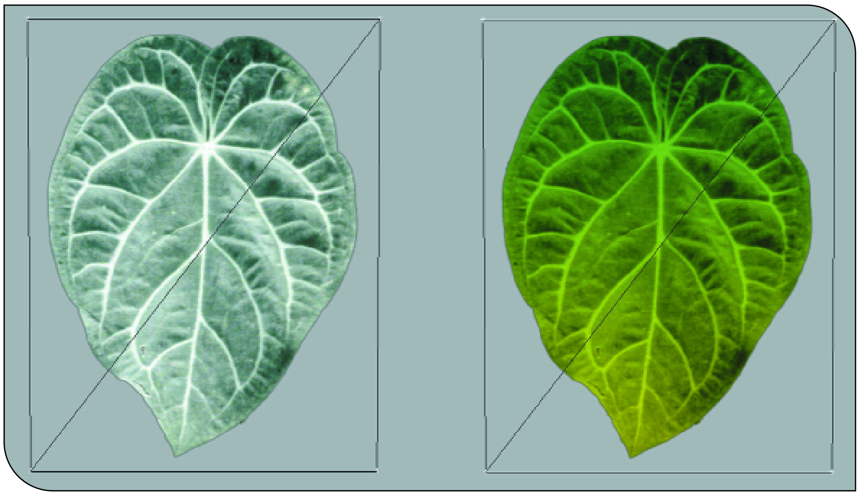
Significantly, many of these worlds are only conclusively established as alien when some form of creature is identified. Think back to when C-3PO and R2-D2 landed their escape pod on Tatooine; as they wandered the desert, the audience could easily have chosen to believe that the sequence was set in Tunisia, for example, until they saw the skeleton of an unfeasibly large beast stretched out along the dunes. Ewoks do the same on Endor, and the Tauntaun lets the audience know that the rebels haven't just set up their base in Alaska. Landscape is important, but it is certainly not the deciding factor when producing an alien world.

Quite often, the game design itself will dictate the kind of world that needs to be created, the physical conditions of the planet, and the state of its inhabitants. Is it an advanced world with highly evolved technology, or is it primitive and primordial? Visually convincing a player that they have journeyed somewhere alien is usually the combined result of well-designed landscapes, creatures, buildings, vehicles, and technology. All of these elements, however, rely on the same kinds of component parts. One of the most important of these is surface.

Surface: Color

Alien worlds, as I discussed previously, may well have much in common with the world we are used to. Rocks, dirt, and sand may all be similar to those we see around us daily. We know from the exploration of our own solar system, as well as the more comprehensive travels of the Starship *Enterprise*, that planets, wherever they are, generally adhere to similar laws of nature. In this respect, when we consider textures for certain elements of an alien world, we can use regular textures as a starting point, with a view to making certain adaptations where





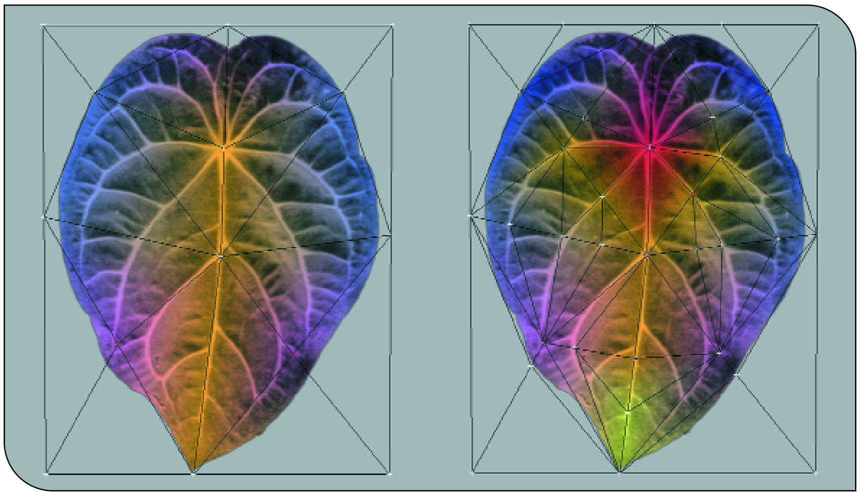


FIGURE 3 (top). Color variation through texture. FIGURE 4 (center). Simple vertex coloring across a quad. FIGURE 5 (bottom). More detailed vertex coloring through added vertices.

necessary. One area that is ripe for a change is that of color.

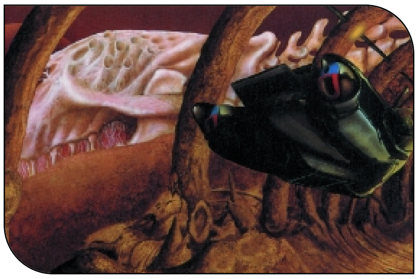
Geological features of a landscape are usually acceptable as alien if they remain within the general ranges of color that we see on Earth. I have, however, argued before that too much brown and gray in a game world fails to take advantage of the colors that we see in nature even when we're restricted to this planet. We have an opportunity to liven up the landscapes of an alien world with some exciting color choices. The trick here is to make sure that a terrain stays clear of becoming cartoony. Bright yellow and orange rocks with highly saturated purple mountains might be ideal as the setting for the Roadrunner to hand out some anvil-shaped pain to Wile E. Coyote, but can easily overstep the mark when attempting to make something more believable.

Again, using our own world as a model, plants and animals show the most vivid range of color variation. A leaf that may look tropical when taken directly from a photograph can begin to look alien when we change its color.

Figure 3 is demonstrating the color changes available through texture manipulation. In that example, I kept the leaf's shape the same and adjusted the color balance in Photoshop to give it a different base color. I then added variation into a separate layer, making sure to follow the leaf's structure so as not to lose too much of its original surface form; I then combined the layers into the final image. Using vertex coloration instead of generating several unique textures can go some way in delivering a wider variation at reduced cost to texture storage, but this technique will produce different, less versatile results.

Figure 4 shows color change through basic vertex coloring across a quad. You can see that mapping this texture onto a quad, using its alpha channel to define the leaf's shape, will only allow four vertices to be given color information, and the effect will therefore be rather diffuse.

Figure 5 shows more complex vertex coloring when more detail is added to the geometry. Adding vertices gives the leaf



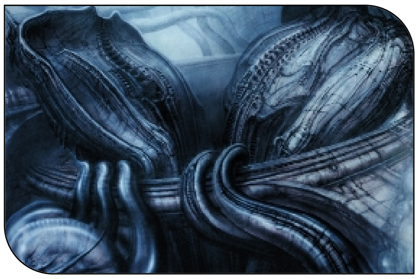


FIGURE 6 (top). Jim Burns' alien organic landscape. FIGURE 7 (bottom). H. R . Giger's bio-mechanical landscape.

more shape in three dimensions and allows an artist to apply a more controlled and versatile set of vertex colors. As always, you need to strike a balance somewhere between an acceptable effect and a wildly overdetailed piece of geometry.

As you can see from the examples I've given, vertex coloring offers artists less control over color distribution than they have when creating a specific texture. It also flattens the texture onto which it is applied as a result of the way that vertex color information and that of the texture below it are combined. With this in mind, you can often obtain the best results when you use vertex coloring to provide general color variation within the geometry, as opposed to complex surface coloration.

Surface Detail

For the last the last few years, artists have had to choose between creating detail as geometry or as part of a texture; this trend has reversed itself in the last year. The next generation of consoles and the most recent additions to the already monstrous lineup of graphics cards have changed the layout of the field on which the game artist does battle. A short while ago, the Army of the Polygon battled regularly with the Forces of Texture, and more often than not was forced into hasty retreat by its inability to support enough detail in the geometry itself. The Cohorts of Texture would sit back smugly, watching the Devotees of the Triangle crumbling under the limited support hardware was able to provide, while they wheeled out a few of their big 512×512 guns just to show off what texture compression allowed them to do.

But a change was inevitable, and suddenly, holding the banner of next generation aloft, the new wave of hardware brought a smile to those who love their polygons.

The question still remains whether it is more economical to use textures or polygons, and what will the impact of either choice be in terms of visual quality (back to the issue of spending budgets where they will do the most good). The difference now is that the overhead for drawing 10,000 extra polygons per frame may well be less than adding a number of extra textures. GPU muscle has concentrated on propelling the marketing-friendly triangle-shifting numbers through the roof, and texture handling hasn't received anywhere near the same attention (please listen to me, Mr. Sony). In addition to this advance, bump mapping is at last beginning to be available as a practical option, which allows detail to be added through texture (a bump map), but which mimics (to some extent) surface features in geometry. As usual, choosing the best approach has to be based on the rigors of your engine and the demands of the particular location being considered.

So the method with which you can add surface detail is now more flexible, but what kinds of surface detail will help build the illusion of an alien world? One direction often taken is exemplified by the well-known fantasy artist Jim Burns (see Figure 6). Among his many trademark approaches to portraying the alien in his paintings, Mr. Burns devotes a lot of time to surface details, implying that the landscape, or objects seen, are made from unfamiliar organic materials.

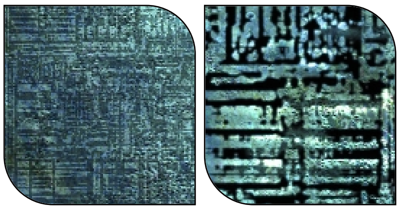


FIGURE 8 (left). A base texture. FIGURE 9 (right). A detail texture.

In contrast, perhaps the most famous creator of all things alien, H. R. Giger (Figure 7), uses surface detail to communicate inorganic interference with organic forms. His "biomechanical" designs have indeed spawned a never-ending stream of imitations that take the chaos and freedom of the natural world and combine it with the order and structure of man-made forms, creating a hybrid style that has become firmly entrenched as synonymous with alien life.

Creating interesting alien surface characteristics, especially for use with objects or areas that are reasonably large, presents the game artist with the familiar challenge of conveying detail at close range while avoiding noticeable repeat in the texture when the player is at a distance. There are numerous ways to avoid this problem, but one that is particularly useful is that of employing layers of appropriate detail, which fade in or out, depending on the distance of the camera from the object in question.

The number of layers you use depends on the constraints of each particular project but will, if allowed to run wild, grind any engine into a pile of smoking silicon. As a rule, two layers are usually sufficient, one that gives a more acceptable large-scale appearance, and the second, to be mapped much tighter, for close-range surface detail. The second layer can be applied so that it has a noticeable repeat, but must be set only to fade in above the base layer once the player is close, so that the repetition is not obtrusive. Depending on the textures you use, the detail layer can be set to remain below a certain level of opacity (60 percent for example), so that larger scale variations in the base layer will continue to provide some changes across the surface. In this case, the detail layer needs to have relatively high levels of contrast (see Figures 8 and 9).

Surface Matters

The factors that contribute to the successful creation of an alien world are many and varied. One area not to be overlooked is that of surfaces, and concentrating appropriate resources on creating surfaces that work can add significantly to the overall feel of an environment. Using what the player knows about our own world as a starting point allows us as artists to contrast the familiar with the unknown, the result being accessible for the player to interpret while hopefully placing them in alien environment that is new and interesting.

Audio Mastering: Taking Game Audio to the Next Level

A few short years ago, the game-playing public practically begged for better sound quality. We have seen some dramatic results stemming from this demand; today's new standards include audio sample rates of at least 22kHz, 16-bit resolution, and some rates as high as 44.1kHz. With this improved CD-quality sound becoming more prevalent, now is a good time to take a hard look at what developers can collectively do to make their sound shine.

Most game companies don't officially address mastering, leaving it instead to whatever the audio person feels is necessary before delivery. That particular mindset was O.K. with the not-so-great playback systems and low sound resolutions of years past. But now that we are doing surround sound for playback on home theater systems, it would behoove us all to give that little extra attention to push game sound over the top. Audio mastering is the key.

What is mastering? Mastering, in its basic form, is taking audio content that has been created over the course of several weeks, smoothing out the bumps, and making it into a cohesive collection. This includes adjusting the perceived loudness of each track and equalizing frequency bands so that once a player (or programmer) has set volume and/or EQ levels, they can concentrate on gameplay and not wonder why one music track is louder than another, or why the speakers are distorting from the bass on another track.

The music industry subscribes fervently to this routine process — music CDs are never released without going through it. There are even professional mastering houses that do nothing but take these types of projects, iron out the wrinkles, and enhance the soundscape.

An absorbing game that totally immerses the player has a better chance of success. Audio tracks that don't stick out like a sore thumb are key to this total immersion. The mastering process can guarantee this actually happens on purpose.

Specific benefits. In this business of tight production schedules, it's not always worth the extra expense in time or money for what some may perceive as a small benefit. But is it really such an imperceptible gain?

Composers and sound designers customarily create their work at the highest resolution possible, normally 44kHz, 16-bit stereo, and downsample to what the project requires. During this conversion, there is always a loss of fidelity. Mastering will offset this loss by reintroducing some high-end "sparkle," usually 3 to 5 decibels in the 8kHz range. This keeps the audio powerful and sonically interesting to the listener.

Music, sound effects, and dialogue require a certain amount of uniformity for each category, each one being delivered at roughly the same volume. With this in mind, the programmer implementing audio can control the levels in code. All background ambiance, for example, could be coded at level 4, dialogue to 7 or 8, and explosions at the maximum level 10. The mastering process is essential to making this happen.

Total immersion requires masking outside distractions. Programmers can set audio levels above the noise of computer fans, spinning hard drives, and speaker hiss. Audio content providers can do their part by ensuring that the loudness of the sound they deliver is above any inherent noise floor and that they don't themselves add noise to the equation. Mastering can maximize sound levels to accomplish this specific responsibility.

Using a professional mastering house or a third-party mastering engineer will afford added bonuses. By having a fresh set of trained ears working on the audio, essential, unbiased adjustments can be made to enhance the sonic experience. Additionally, high-end mastering equipment does more than just even out the EQs and compress the various audio tracks to comparable levels. Good mastering preserves the stereo field, enhances imaging, and increases dynamic range, while also evening out the program.

So what does it take? Mastering can be accomplished by the content provider, before final delivery, by taking a step back and seriously evaluating the needs of the audio. The goals of enhanced audio can be easily accomplished by running the audio through hardware processors or software counterparts such as TC MasterX, Waves Renaissance or Ultramaximizer, or T-Racks 24.

Sending a game project's audio to a professional mastering facility is also a great choice. An hour's worth of music, for example, can be processed in roughly three to four hours. An average rate of $250 per hour means your game audio can have all the benefits and enhancements for around $1,000 — worth the relatively modest investment.

The audience is listening. Audio mastering is slowly working its way into the game audio scene. In the past, understandably, it wasn't important. But as playback systems, processing speed, and storage capabilities continue to improve, it is critical that our attention be given to making game audio the best it can be.

My thanks to Keith Arem, Darryl Duncan, Tim Larkin, Tommy Tallarico, Mark Temple, and Chance Thomas for sharing their wisdom for this column.

AARON MARKS | *Aaron (aaron@onyourmarkmusic.com), composer, sound designer, and author of* The Complete Guide to Game Audio *(CMP Books), is the humble proprietor of On Your Mark Music Productions (www.onyourmarkmusic.com). He is currently hard at work on game projects for Vivendi/ Universal, Flipside.com, and Enemy Technology.*

The 400 Project Continued: Providing Parallel Challenges

Welcome to another game design rule from the 400 Project. This month's rule has wide applications to most game designs.

The Rule:
Provide Parallel Challenges with Mutual Assistance

When presenting the player with a challenge — a monster to kill, a puzzle to solve, a city to capture — provide several such challenges and set it up so accomplishing one challenge makes it a little easier to accomplish the others (that's the mutual-assistance component). Setting up these parallel challenges on many levels of scale of the game, from the ultimate goal down to the small, short-term steps, is also very effective. Doing so eliminates bottlenecks and makes the game accessible to a wider range of players.

The rule's domain. This is a basic rule of game design, applying to all games directly.

Rules that it trumps. There are as yet no rules in our collection that this one trumps.

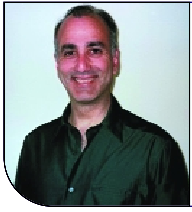
Rules that it is trumped by. This rule is trumped by "Provide Clear Short-Term Goals," our rule from last month's column. It is important not to let parallel challenges confuse the player about what must be accomplished. But the two rules can easily co-exist in harmony if the parallel challenges are clear steps necessary for a larger goal. In SMALL SOLDIERS: SQUAD COMMANDER from Dreamworks Interactive, we broke the single goal of freeing an ally into two subgoals of "free his feet" and "free his hands," each of which became an entire mission that could be accomplished in either order.

Examples from games. This rule is used effectively in many classic, successful games. Sid Meier's CIVILIZATION series of games is practically a case study of extensive use of this rule, nested recursively on many levels. On the highest level, the objective is to win the game — but this can be done in the original game by conquering all the other civilizations in the world, or by being the first to send a starship to Alpha Centauri. If a player focuses on conquest, it still can help to pay attention to building technology that leads to the starship victory, as this technology provides advantages in conquest as well. If the player focuses on the starship victory, limited conquest of neighboring civilizations can provide the resources needed to achieve it. The recent CIVILIZATION III adds various parallel diplomatic and cultural avenues to win the game. But deeper down in the game, the rule is applied even more directly. At any point there are challenges of improving individual cities, building the military, accumulating wealth, engaging in diplomacy, and researching new technology. Moreover, success in any of these can make it easier to achieve the others.

DIABLO II is another fine example. Unlike many other, less successful games, players are never left with a single bottleneck challenge that must be surpassed by the frustration of repeated vain attempts. Completing one of several available quests makes your character incrementally stronger by gaining a new level, or wins better armor or magic, making the

In CIVILIZATION III, diplomacy is one of several parallel paths to victory in the game, and success in other CIVILIZATION goals.

other quests slightly easier. Even the apparent bottlenecks of tough boss monsters at the end of each act of the game are really parallel challenges with mutual assistance. You are required to fight the boss to progress forward, but you can always go back and repeat earlier quests, allowing you to face the boss with a higher-level, better-prepared character. This structure was effective in making DIABLO II into a multi-million-unit seller, because it has made the game accessible to a wide range of skill levels. A very experienced player can zoom through and fight Andariel, the first-act end boss, with a character that has only achieved level 15 and accordingly must be handled masterfully, providing a tough and exciting challenge. Novice players can stay in their comfort zone, taking their time to reach Andariel, raising their character to level 20 or higher, and gaining new weapons and armor. For them, Andariel will still be an exciting challenge that they'll vanquish only after a satisfying fight, despite their more modest game-playing skills.


NOAH FALSTEIN | *Noah is a 22-year veteran of the game industry. You can find a list of his credits and other information at www.theinspiracy.com. If you're an experienced game designer interested in contributing to The 400 Project, please e-mail Noah at noah@theinspiracy.com (include your game design background) for more information about how to submit rules.*

Return

Rendering the Great
Fast Occlusion Culling for Outdoor

Rendering engines used in today's game titles utilize various techniques for hidden surface removal (HSR), different techniques being suitable for different game genres. For example, action games played in closed-in locations such as rooms, caves, and tunnels need an engine allowing fast rendering of a few hundred polygons, a high frame rate, and a high level of details to impress players. Conversely, a game that is taking place outdoors requires quite a different approach. Let's discuss appropriate approaches for the latter.

Not too long ago, games ran in software mode only, without the help of 3D accelerators. With the CPU doing all the work, engines rendered as few pixels as possible, typically with BSP-based scenes and BSP rendering.

Moving Games Outdoors

With the advent of 3D accelerators and the invention of the depth-buffer (or Z-buffer), the strict sorting of polygons slowly faded out of engines, and software engineers started trying out different techniques. Game designers wanted to move their worlds outdoors, and new graphics hardware made such designs possible.

As graphics hardware power increases, however, so do requirements for game content and high-quality graphics, creating room to waste processing power with inefficient usage of computing resources.

Let's discuss one technique for hidden surface removal usable in 3D engines, developed while creating some of the games I've worked on. The technique, utilizing object occlusion, is for outdoor rendering. The fundamental entities working for us will be so-called occluders, and we'll come to them soon. First, some rendering theory.

Figure 1 shows a rendering pipeline using the technique presented here. This is a higher-level look at the rendering process, without going into details about actual polygon rendering.

MICHAL BACIK | *Michal is the lead programmer at Lonely Cat Games, a small company based in the Czech Republic. Previously he was lead programmer for* HIDDEN & DANGEROUS*, a World War II title released two years ago. Recently he finished work on* H&D DELUXE*, an improved version of the original game. Michal can be reached at michal@lonelycatgames.com.*

The Scene Hierarchy Tree

One of the first steps toward optimized rendering is keeping objects in a scene organized. Most commercial modeling packages and 3D engines use a scene hierarchy, and for good reason: used smartly, it allows for fast rejection of entire hierarchy branches based on bounding volume tests, and may also accelerate collision testing.

A scene hierarchy tree is a collection of objects (visuals as well as nonvisuals) in an organized fashion. In a tree structure, a scene has its root object, which may have children objects linked to it, and these child objects may have other objects linked to them. The objects may be visuals (characters, terrain, items in the game), as well as nonvisual objects (such as 3D-positioned lights, sounds, dummies, and other helper objects).

Balancing the Tree

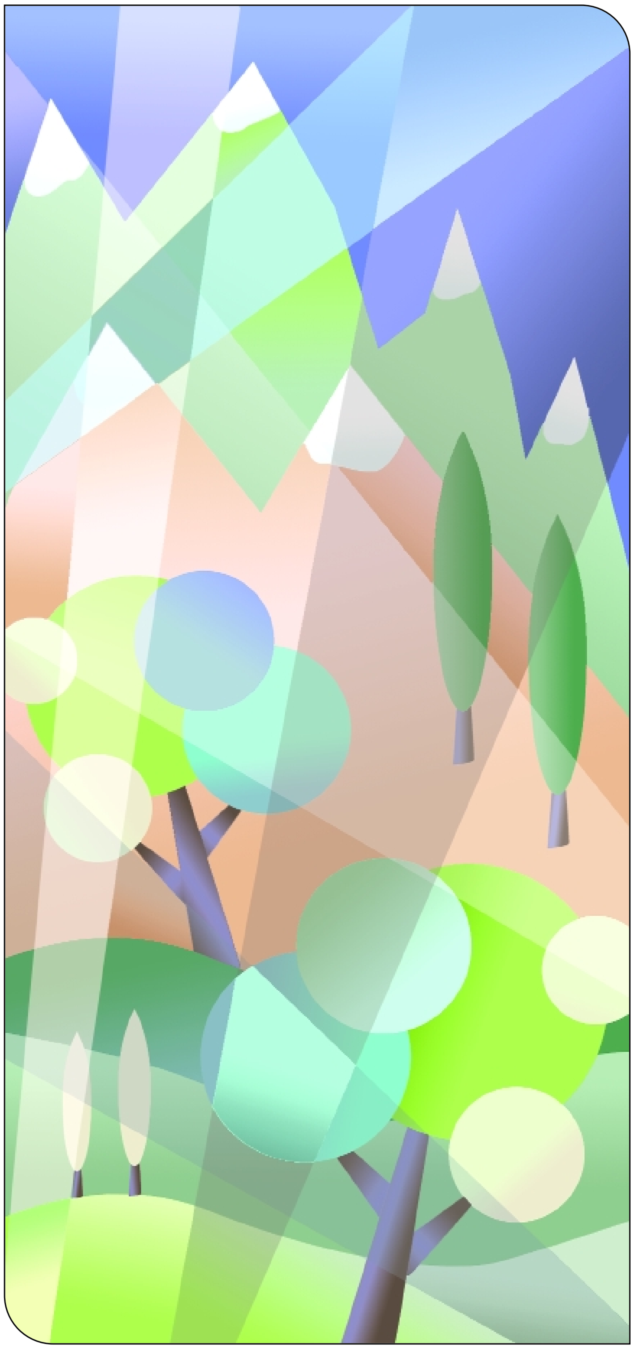
When creating the hierarchy tree in a commercial 3D graphics package or in-game editor, keep the tree balanced. For example, by linking all your objects to the root level, you lose the benefits of having a hierarchy, because all your objects are in a single list and any processing will just treat them as an array. Grouping objects logically is a big step forward.

Hierarchical object linking is usually based on some logical assumptions. For preprocessing purposes, keep multiple adjacent objects together, linked to one parent object. Object linking is also done for other purposes, such as inheriting the parent's position, rotation, and scale. This is achieved by using matrices and matrix concatenation.

Bounding Volumes

A bounding volume is a simple geometrical object roughly representing the volume of a real object's geometry. It's as small as possible while still enclosing all vertices of the object. The most suitable geometric objects for bounding volumes are spheres and boxes. For the techniques presented in this article, I

Outdoors: Environments



recommend using both types and defining a structure representing the bounding volume:

```
struct S_bounding_volume{
   struct{
      struct{
         float x, y, z;
      } min, max;
   } box;
   struct{
      struct{
         float x, y, z;
      } pos;
      float radius;
   } sphere;
};
```

You're probably already using an HSR technique utilizing bounding boxes or spheres, so why use a combination of both? The answer is simple: speed. While bounding spheres allow for very fast collision detection using a simple distance test, the volume it encloses is often much greater than the actual object it represents (Figure 2). Too often, then, we consider an object to be on-screen when none of its vertices actually would be.

On the other hand, a bounding box is a closer match to the shape of an object, but tests with boxes are slower. Usually we do a two-pass collision/intersection test, using a bounding sphere in a first pass and oriented bounding box (OBB) in the second pass. Because the first test rejects most invisible or non-clipped objects, the chances that a second test will never be executed are high. At the same time, performing the bounding box test yields more accurate results, leading to fewer objects being rendered.

This mixed bounding volume is also suitable for other purposes, such as collision detection or physics.

Node Volumes

Every visual object in a scene hierarchy should have an associated bounding volume, so that with just a few math operations, we're able to say if each object is visible or not. As I mentioned previously, it wouldn't be very efficient to test all the scene objects in each rendered frame. For example, in a racing game with 20 cars, you'd end up testing all the vehicle's objects (wheels, doors, driver, driver's fingers, and on and on) 20 times, which could easily make for upwards of 25 objects on each car multiplied by 20 cars. Why perform 25 tests when a single test for each car can accomplish the same result?

With complex models or models used in a scene multiple times, finding a way to skip the processing of an entire model or models if they're out of view would be highly desirable. For this reason, we'll introduce a node bounding volume. This is the same bounding structure defined above, but it doesn't enclose vertices; rather it encloses all bounding volumes of a group of objects, the child objects of the node (objects, models, and so on).

Building and Maintaining Volumes

Assuming you have static geometry, calculating a visual object's extent is done once before starting rendering. The bounding box is axis-aligned (AABB) in the local coordinates of an object, defined by two extreme points of the box. Anytime a bounding box is used in computations, it must be transformed into world coordinates using the object's transformation matrix, and thus changed to an oriented bounding box. Because OBBs cannot be specified by just two corner points, we'll need to extract the two corner points of the AABB into the eight corner points of the OBB, and transform all these points to world coordinates with the object's transformation matrix (the same one that will be used to transform the object's vertices to world coordinates). The bounding sphere is also in local coordinates, and must be transformed (position) and scaled (radius) to world coordinates before being used in computations.

The situation with node bounding volumes is a bit more difficult. Because the position of objects may change in real time, this bounding volume must be computed at run time whenever any object in the group moves. The best method is a lazy evaluation programming technique — in other words, computing the value when it's needed. You may implement a system of invalidation of a node's bounding volume when the child's matrix changes due to position, rotation, or scale. This system is harder to implement and debug, but it's critical for fast 3D culling, both rendering and collision testing.

By measurements I've made in our 3D system, the dynamic bounding volume update takes no more than 1 to 2 percent of total CPU time when the game is running.

Convex Hull Computation

Because it's very easy to detect collision against it, a convex hull is another basic geometry object used in occlusion testing. During occlusion testing, we'll detect a collision of a 3D point with a hull and a sphere with a hull. Since we must detect how the bounding volume of an object collides with viewing volumes (screen frustum and occlusion frustum), hidden-surface removal has much in common with collision detection.

A hull is defined as a set of planes that forms the hull, with their normals pointing away from the hull. Any point in space is a part of the hull if it lies behind all the planes forming the hull. For our purposes, we'll also use an open hull, which is a hull that represents an open 3D volume.

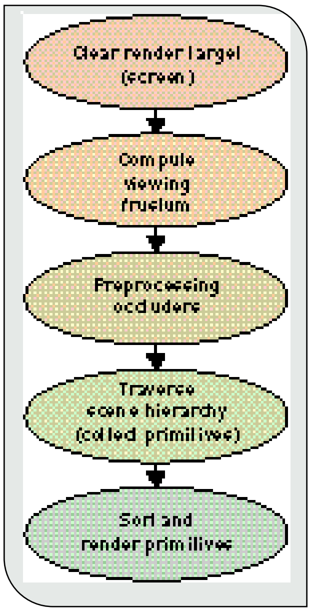

Clear render target (screen)
Compute viewing frustum
Preprocessing occluders
Traverse scene hierarchy (culled primitives)
Sort and render primitives

FIGURE 1. Preprocessing part of the rendering pipeline.

LISTING 1. Checks if a 3D point is inside a convex hull. A convex hull is defined as a set of planes with normals pointing away from the hull. This function uses the `S_vector` class for a 3D point, the `S_plane` class for a plane in 3D space, and a member function `S_vector::DistanceToPlane(const S_plane&) const` that determines the distance from a 3D point to a plane.

```
bool IsPointInHull(const S_vector &v, const vector<S_plane>
&planes){

   for(int i = planes.size(); i--; ){
      float d = v.DistanceToPlane(planes[i]);
      if(d >= 0.0f)
         return false;
   }
   return true;
}
```

All the information we need to compute the convex hull is a set of 3D points. During the computation, we'll remove redundant points, which are inside the hull and do not lie on the skeleton of the hull. We'll also need to compute the edge faces of the convex hull; these faces are not necessarily triangles, as they are in 3D meshes.

We'll utilize planes of these edge faces for our occlusion computations: for example, a fast check of whether a 3D point is inside of a convex hull. If a point in space is behind all the edge faces of the hull (assuming the planes' normals point out from the hull), then it is inside of the hull (Listing 1).

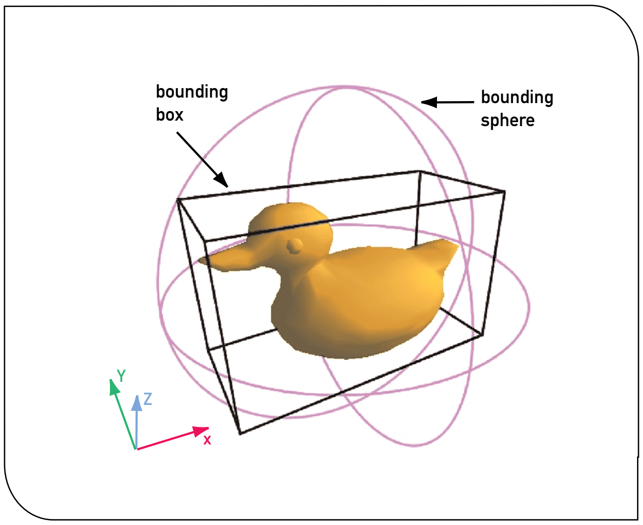


FIGURE 2. Visual object with bounding box and sphere. Notice that the sphere is a bit larger than the box.

Several algorithms exist for computing a convex hull from a set of points. Some commonly used ones include incremental, gift-wrapping, divide-and-conquer, and quick-hull. There is a nice Java applet demonstrating techniques of building convex hulls that is a good starting point for choosing and implementing the algorithm. It is available (with kind permission of its author) at www.cse.unsw.edu.au/~lambert/java/3d/hull.html.

Speaking from my own experience, writing code for building convex hulls is quite a difficult task. Even when you implement the code properly, you'll encounter problems with float-rounding errors (using doubles won't solve anything). Regardless of which algorithm you choose, with some point sets you'll end up with hulls that are invalid after final validity checks are performed. Having multiple points placed close together on a plane is a common source of problems.

After I struggled for months with code that computed it wrong, then spending another month writing my own convex-hull computation code, I finally switched to Qhull, a freeware package that utilizes the quick-hull algorithm. It's available at www.geom.umn.edu/software/qhull.

Although the QHull library is a robust, platform-independent package that can do many additional tasks, we will only be needing it to compute the convex hull using our structures. It handles rounding problems by joggling points; if computation fails, it shifts points randomly by a small value and recomputes until it gets a proper hull.

If you decide to use this or any other available package, be prepared to spend a day or two reading its documentation and writing code to call it; you'll save a month otherwise spent writing and debugging your own system.

The final result we need after computation is a set of filtered points that form a skeleton of the hull, and a set of faces. Following is an example of what we get (faces are kept as indices in the point set):

```
struct S_vector{
   float x, y, z;
};
struct S_face{
   int num_points;
   unsigned short *indices;
};
```

Now we use the help of C++ STL `vector` class for storing our vertices and faces:

```
std::vector<S_vector> hull_points;
std::vector<S_face> hull_faces;
```

Note that when inserting into `vector`, a copy constructor of the class being inserted is called. Make sure you have implemented a copy constructor of `S_face` so that memory for `indices` is properly allocated and freed.

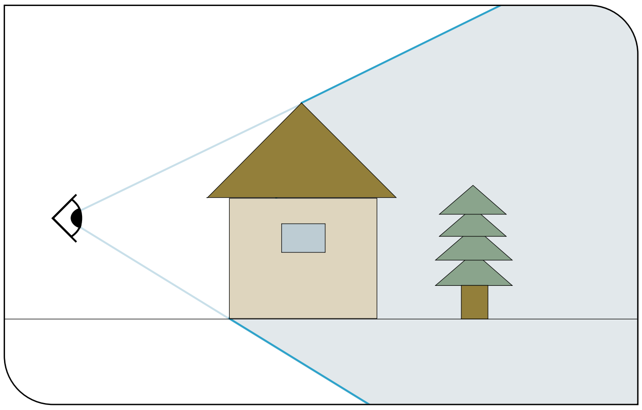


FIGURE 3. An example of occlusion in the real world. The house occludes the view onto the tree; the shaded area represents the volume that the viewer cannot see.

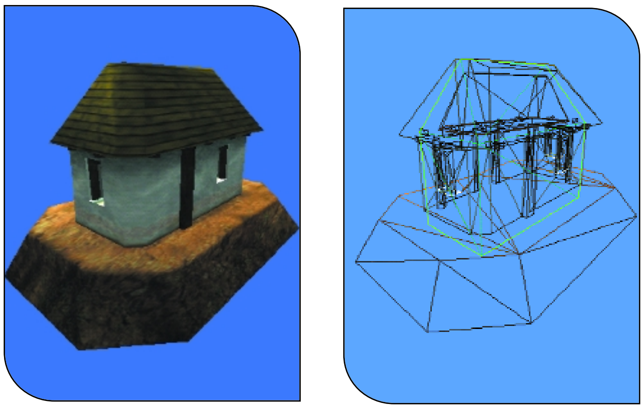


FIGURE 4 (left). A house, suitable for occluding other objects in 3D scene. FIGURE 5 (right). The occluder object (drawn in green wireframe), representing the simplified shape of the house.

The View Frustum

A viewing frustum is a 3D volume. For practical reasons, we can treat it as a convex hull, simplifying further computations. The viewing frustum is typically a cut pyramid (if a projection transformation is used) or a cube (if an orthogonal transformation is used). This article assumes that projection transformation is used, and many times the camera's position will form one of the points of the frustum's hull.

Making Sense of Occluders

Occluders in the real world may be thought of as objects which occlude (or obstruct) your view of other objects behind them (Figure 3). Take an example of an occluder — a building, a hill, or a car — all these things occlude your view to objects physically located behind them. Transparent or translucent objects are not good view occluders, because as light rays

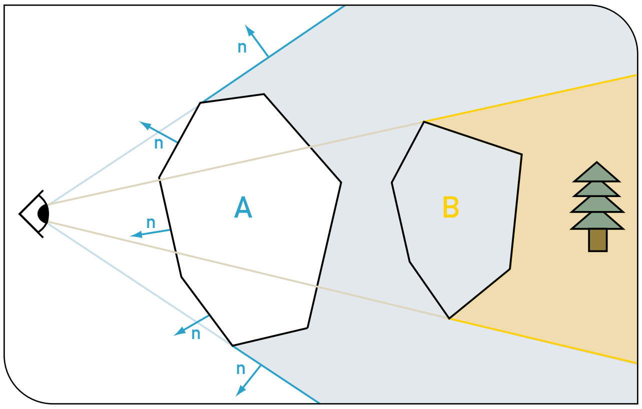


FIGURE 6. Image showing an occluder hidden by another occluder. Any visual occluded by the occluder B is also occluded by occluder A, so we include only occluder A in our list of occluders we test against.

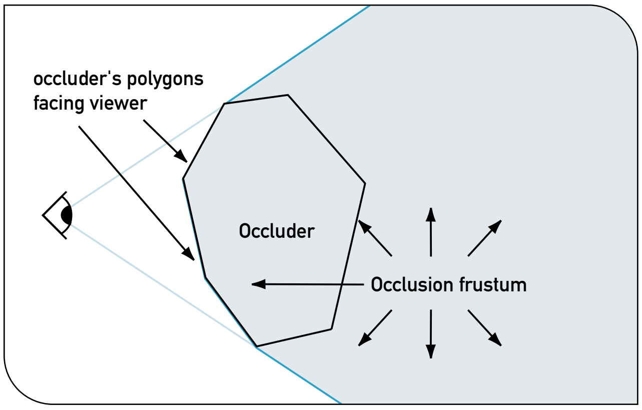


FIGURE 7. An Oocclusion frustum built from an occluder, using current viewer position.

pass through the material, so we'll ignore transparent objects as occluders in 3D rendering.

Objects in the real world consist of atoms and molecules, and pretty much every atom can either be occluded by another atom or not (in which case a viewer can see it). In computer graphics, however, objects are built from vertices and polygons, and these vertices and polygons are usually grouped into primitives, which are rendered together in order to achieve good graphics throughput. Our task consists of rejecting as many of these primitives as possible in the early (preprocessing) phase of our pipeline, without affecting the viewer's experience by rejecting objects that should be rendered.

In practice, this means finding which objects are fully occluded by other objects and rejecting these occluded objects from any further processing. A solid object, sufficiently big to be worth the additional computations associated with occlusion testing, is an ideal occluder.

In an ideal 3D engine, we could detect occlusion of even the smallest primitive behind any object in a scene. In reality, however, we must find some algorithm that allows fast detection of occlusion for a sufficient number of potentially visible primitives. For that reason, we'll simplify the occlusion volumes to convex hulls.

Convex hulls allow for sufficient approximation of a 3D object in most cases. When it is not possible to represent the shape of an object with a convex hull, you can use more hulls to accomplish the task. The occlusion hull doesn't need to copy the exact shape of visual object it works with. In many cases, an occluder may consist of many fewer faces than the visual primitive itself, roughly copying the shape of the visual (Figures 4 and 5). The rule to keep in mind here is that an occluder's shape shouldn't be bigger than the shape of visuals it represents; otherwise your engine will end up rejecting primitives that should be rendered, resulting in an ugly graphical artifact.

The Occluder's Place in the Pipeline

In determining HSR, occluders should be processed first. Because the position of a camera (viewer) changes constantly in 3D games, so does the occlusion frustum, the 3D volume cast by the occluder. Our task is to compute the occlusion volume at the beginning of rendering from a particular camera view. (If you render multiple views in a single frame, a mirror for example, this step must be done for each rendered view.) After this preprocessing step, you should collect all occluders on the screen, against which you'll test other potentially visible primitives.

Some optimization tips: Minimize the number of occluders you include in the test list, done by determining if a particular occluder is occluded by another occluder. Figure 6 shows this possibility. Also, don't consider occluders that may potentially hide only a small amount of primitives. You should reject occluders that occupy a small area of screen space.

Once we have a list of on-screen occluders, we can move on to the next preprocessing step: traversing the scene hierarchy tree and using the list of occluders to check if a particular object is visible.

Building Occlusion Volumes

Let's have a closer look at the information needed in order to detect whether an object is occluded. Looking closer at the occlusion volume, we see that occlusion volume is actually another kind of convex hull, expanded from the viewpoint into infinity. The occlusion volume is built from all of the occluder's polygons facing the camera, and from contour edges (as seen from the camera) expanded away from the camera (Figure 7).

Actually, this volume is open — there's no back plane that would cap the volume — because the occluder hides everything behind it into infinity. And any plane we save will speed up further computations.

To build contours from a convex hull, we use a simple algorithm utilizing the fact that each edge in a convex hull connects exactly two faces. The algorithm is this:

1. Iterate through all polygons, and detect whether a polygon faces the viewer. (To detect whether a polygon faces the viewer, use the dot product of the polygon's normal and direction to any of the polygon's vertices. When this is less than 0, the polygon faces the viewer.)
2. If the polygon faces viewer, do the following for all its edges:
   If the edge is already in the edge list, remove the edge from the list.
   Otherwise, add the edge into the list.

After this, we should have collected all the edges forming the occluder's contour, as seen from the viewer's position. Once you've got it, it's time to build the occlusion frustum itself, as shown in Figure 7 (note that this figure shows a 2D view of the situation). The frustum is a set of planes defining a volume being occluded. The property of this occlusion volume is that any point lying behind all planes of this volume is inside of the volume, and thus is occluded. So in order to define an occlusion volume, we just need a set of planes forming the occlusion volume.

Looking closer, we can see that the frustum is made of all of the occluder's polygons facing the viewer, and from new planes made of edges and the viewer's position. So we will do the following:

1. Add planes of all facing polygons of the occluder.
2. Construct planes from two points of each edge and the viewer's position.

If you've gotten this far and it's all working for you, there's one useful optimization to implement at this point. It lies in minimizing the number of facing planes (which will speed up intersection detection). You may achieve this by collapsing all the facing planes into a single plane, with a normal made of the weighted sum of all the facing planes. Each participating normal is weighted by the area of its polygon. Finally, the length of the computed normal is made unit-length. The *d* part of this plane is computed using the farthest contour point. Occlusion testing will work well without this optimization, but implementing it will speed up further computations without loss of accuracy.

Detecting Occlusion

To detect if an object is occluded, we will utilize the object's bounding volume. To find out if the object is inside of the occlusion frustum, we'll make a simple test to check if its bounding sphere is inside of the frustum. Figure 8 shows possible situations that may arise. Here we see that only sphere C passed the test and is fully occluded. Listing 3 shows the function that may be used for such a computation.

This process detects whether a bounding sphere is occluded. It's fast, but not as accurate as detection using bounding boxes. With this technique, some objects may be detected as visible after the bounding sphere test succeeds, but their bounding box is still fully occluded, so we would end up rendering them even though they're fully occluded. Figure 9 illustrates this possibility.

Detecting if the bounding box is inside the occlusion frustum is another very simple task: detect if all eight corner points of the bounding box are inside of the frustum (Listing 2). Note that we use oriented bounding boxes in world coordinates, so we must transform local AABBs to world OBBs (as explained previously). If any vertex is outside of the volume, the box is not occluded. This test can take eight times more dot products than the sphere test, so it is less efficient. Ideally you would use it only when you detect that the center of the bounding sphere is inside the occlusion frustum but the sphere is still not occluded. This minimizes the chances of wasting time checking box-versus-frustum collision, at the same time getting more accurate occlusion tests, resulting in fewer objects being rendered.

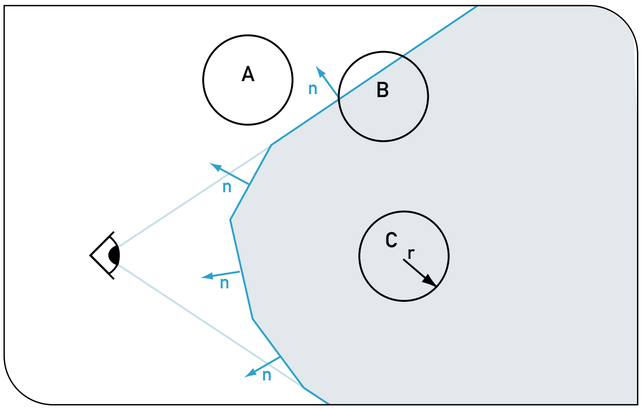


FIGURE 8. Sphere A is outside of at least one plane. Sphere B is inside all planes, but its radius is greater than the distance from one of the planes. Sphere C is behind all planes and a sufficient distance from all planes.

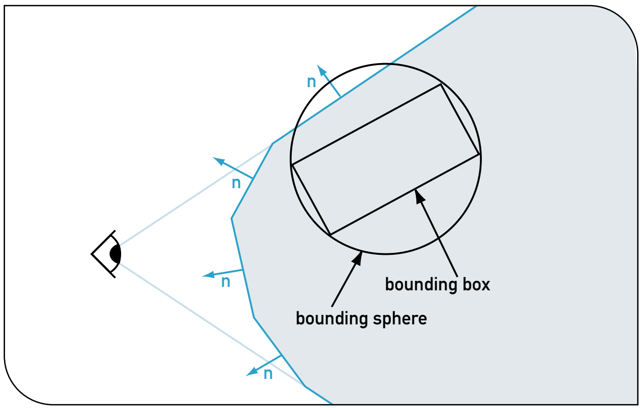


FIGURE 9. A case when the bounding sphere is not fully occluded but the bounding box is, thus the object should not be rendered.

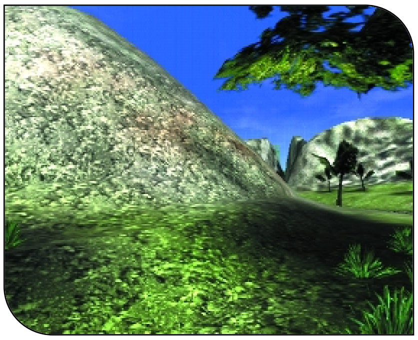
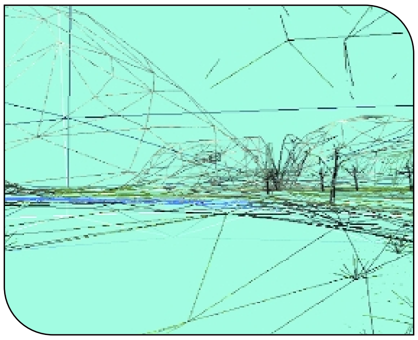
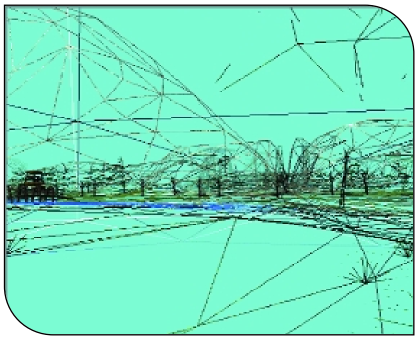




FIGURE 10A (left). The original in-game screenshot. FIGURE 10B (middle). The same screenshot in wireframe mode, with 7,000 triangles rendered at 50 fps. FIGURE 10C (right). Occlusion is switched off. The number of rendered triangles rose to 17,300, the frame rate dropped to 20 frames per second.

Editing Support

Computing the occlusion volume and detecting whether an object is occluded is half of the work that needs to be done. Another task is finding a way to edit occluders comfortably in an evolving 3D scene during development. It may be done in several ways; I'll discuss some of them here.

Editing occlusion volumes in a modeling package. This is a convenient way of starting with occluders quickly and testing functionality. You may use 3DS Max, Maya, or whatever modeling program you prefer, to build an occluder skeleton from vertices and faces and import that into your engine.

Integrating an occluder editor into your game editor. This is a harder way to do it, but it's preferable over editing in a modeling package. The geometry inside occluders may change, and occluders must match geometry in order to be effective, so an "edit and see" approach is the best bet here.

Because an occluder is simply a convex hull, once you've implemented and tuned your convex hull code, you can call it with a set of points and you've got it.

Figures 10a–c show occlusion techniques in a real game project, including numbers about rendered triangles. Note that the terrain is specially modeled so that occlusion is efficient; it contains high hills that are suitable for occluding objects behind them. But the areas of possibility for occluder utilization are much wider: cities with plenty of buildings, fences (solid, not transparent), and so on.

Finally, fine-tune editing support by visually displaying occluders in edit mode while designing your missions or modifying the occluder's shape in the editor.

Take It Outside

This article provides some insight into rendering possibilities for outdoor scenes, and shows some new directions for optimizations. The model described here has worked — and continues to work — for games I have been working on (used with other HSR techniques, such as sectors and portals) and has undergone many changes to get to this point. As we continue to improve our methods for rendering all kinds of environments in games, soon there will be no boundaries as to what kinds of worlds we can provide players.

LISTING 2. Detects if a set of all points is inside of the viewing frustum. This is just a variation of Listing 1, checking multiple points instead of one.

```
bool AreAllVerticesInVF(const vector<S_plane> &planes, const
S_vector *verts, int num_verts){

   for(int i = 0; i < num_verts; i++){
      for(int j = planes.size(); j--; ){
         float d = planes[j].d + planes[j].normal.Dot(verts[i]);
         if(d >= 0.0f)
            return false;
      }
   }
   return true;
}
```

LISTING 3. Detects if a sphere is inside the frustum (convex hull). Notice that this function also computes clipping flag, which is set whenever a sphere collides with one hull's plane (that is, it is neither fully inside nor fully outside of the volume).

```
bool IsSphereInFrustum(const vector<S_plane> &planes,
   const S_vector &sphere_pos, float sphere_radius, bool &clip){

   clip = false;
   for(int i = planes.size(); i--; ){
      float d = planes[i].normal.Dot(sphere_pos) + planes[i].d;
      if(d >= sphere_radius)
         return false;
      if(d >= -sphere_radius)
         clip = true;
   }
   return true;
}
```

Return

Integrating Your Game Engine into the Art Pipeline, Part 2: Integrating with Maya

Many art pipelines are driven by the old truism "The faster you can see the art in-game, the faster you can decide to change it." However, this is based on the game engine being the last stage in the pipeline, and the first time artists see their work rendered in-game comes after they've sent it off to be converted into a format the game can understand, and then placed into the game itself. Unfortunately, by the time they see their work or get art review feedback, they're already working on something new. This system then requires time to go back and fix up or redo the old art and put it back through the long, arduous process of getting it in-game.

Enabling artists to get art into the game as soon as possible, and in near real time, can allow them to iterate on the in-game look of an asset before involving others and can speed reviews of in-game assets. By integrating the game engine directly into Discreet's 3DS Max or Alias|Wavefront's Maya (Figure 1), the artist doesn't even have to leave the modeling environment to check the in-game look of a particular model.

Last month, in the first part of this series ("Integrating Your Game Engine into the Art Pipeline, Part 1: Integrating with 3DS Max"), I discussed the issues that have to be solved in order to convert a stand-alone game engine into a Dynamic Link Library (DLL) that forms the basis for a plug-in for Max or Maya. I then identified the issues associated with implementing a game viewer as an extended viewport plug-in in 3DS Max 4, and the trade-offs that need to be taken into account when exchanging data with Max or attempting to get mouse and keyboard input to the game viewport.

In this concluding article, I'll recap the issues and problems with converting a stand-alone Direct3D-based game engine first into a DLL. Then I'll develop it into a plug-in for Maya 4. I'll also discuss the trade-offs between creating an in-game viewer for Max or Maya and then briefly highlight the changes and differences required to create the same functionality for a game engine that utilizes OpenGL instead of Direct3D.

Throughout this article, references to Max and Maya are completely interchangeable except where the two are being compared, or specific comments are made about Maya (or Max) features and programming.

Getting Ready to Plug In

The first step to getting a stand-alone game engine to run as a plug-in for Maya (or Max) is to convert the engine into a DLL. To demonstrate the changes that need to be made to convert the game engine into a DLL, the Direct3D sample program Skinned Mesh was converted first into a DLL (Figure 2), then into a full-fledged plug-in (Figure 1). Making this a two-step process allows us first to handle the problems created by converting the game engine into a DLL, and then separately handle the problems created by integrating the game engine into Maya itself. All source code for this series is available from the *Game Developer* web site at www.gdmag.com.

When converting a stand-alone game engine into a DLL, chief among the issues to be solved is making sure that global variables and values are set to known states when the game engine shuts down. This is because now that the game is running as a DLL, it can be loaded once and then started and shut down any number times. Since the global variables are only initialized during the load process, the game will work correctly the first time but likely either leak memory or crash outright on subsequent runs.

Converting to a DLL also requires a new way to drive the game update loop. When the game was an executable, the `WinMain` function drove the update, but the game will now need to supply its own driving function in the DLL. The easiest method is to create a thread that will drive the game update, although a timer could be used as well.

Since the game will now be driven by two threads — one it created itself for the update and one from the calling application — care must be taken to make sure the game doesn't try to shut down while rendering or resize the rendering window in the middle of an update. This can be accomplished by

HERB MARSELAS | *Herb is a game programmer at Microsoft's Ensemble Studios, where he is working on the upcoming* AGE OF MYTHOLOGY. *He can be reached at herbm@microsoft.com all hours of the day and night.*

placing critical sections around code that can be executed from both threads. It may also be required to inform the Direct3D device that it's being run from multiple threads, but this can be avoided if the developer is vigilant with his or her critical-section handling.

Although individual game engines may encounter other problems, two other common problems are window and input handling. Since the game will now be running in a child window, the window style needs to be changed to reflect this. Running in a child window may also require changes to mouse positioning, as well as to mouse and keyboard input, because even while the application may have input focus, the game window may not. A full discussion of these issues can be found in last month's article.

With the game successfully running as a DLL, it's now time to take it to the next level and convert it into a Maya plug-in.

Plugging into Maya

Converting a DLL into a Maya plug-in requires two simple steps. The first of these is to change the extension of the DLL from .DLL to .MLL and place it in the AW\Maya4.0\bin\plug-ins folder. This allows Maya to find the plug-in when it's time to load it. As an aside, for simplicity and to eliminate any confusion with the SkinnedMeshDLL and the SkinnedMesh Max plug-in, I renamed the Maya plug-in d3dview.MLL.

The second step is the addition of two functions that are used to initialize the plug-in when it is loaded and shut down the plug-in when it is unloaded. These are called `initializePlugin` and `uninitializePlugin`, respectively (Listing 1). The `initializePlugin` function registers two new commands, called `d3dviewport` and `gameview`. Registering these commands will allow them to be called from Maya's Embedded Language (MEL).

Declaring these new commands is a straightforward process. New commands are derived from the `MPxCommand` class (Listing 2). The derived class then implements its own versions of the static cre-

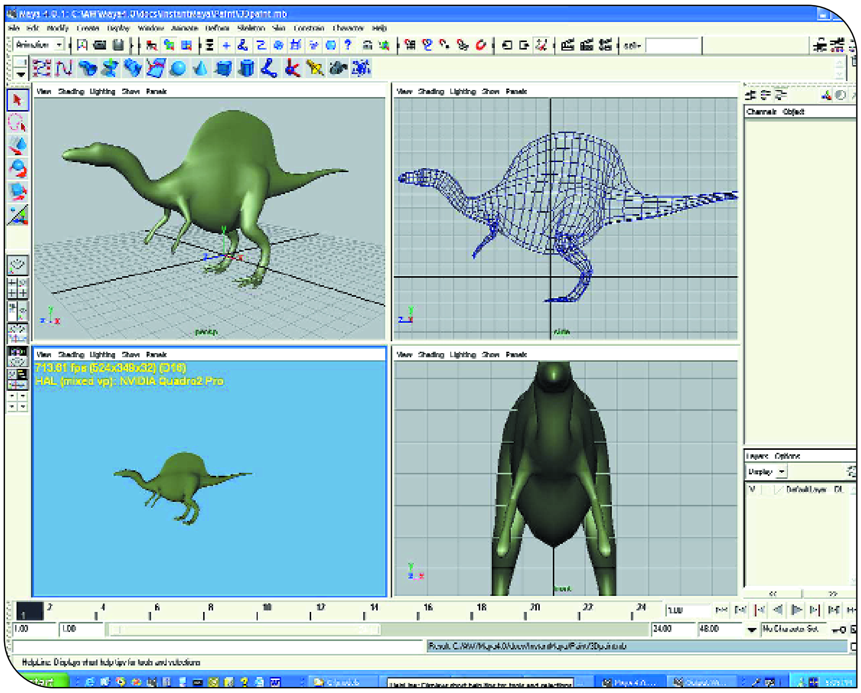

713.61 fps (524x348x32) (D16)
HAL (mixed vp): NVIDIA Quadro2 Pro

FIGURE 1. Direct3D viewer running in the lower-left panel of Maya.

LISTING 1. Registering MEL commands in Maya is a very simple process. `initializePlugin` registers two new MEL commands `d3dviewpoint` and `gameview`. `uninitializePlugin` deregisters the commands when the plugin is unloaded.

```
MStatus initializePlugin( MObject obj )
{
   // d3dviewport command

   MFnPlugin d3dplugin( obj, "GameDevMag", "1.0", "Any" );
   d3dplugin.registerCommand( "d3dviewport", CD3DViewport::creator );

   // gameview command

   MFnPlugin gameplugin( obj, "GameDevMag", "1.0", "Any" );
   gameplugin.registerCommand( "gameview", CGameView::creator );

   // all done

   return MS::kSuccess;
}

MStatus uninitializePlugin( MObject obj )
{
    MFnPlugin d3dplugin( obj );
    d3dplugin.deregisterCommand( "d3dviewport" );

    MFnPlugin gameplugin( obj );
    gameplugin.deregisterCommand( "gameview" );

    return MS::kSuccess;
}
```

`ator` method, which acts like a very simple class factory to instantiate instances of the class, and the `doIt` method, which is called when MEL executes the command. The new command is then registered and the class factory passed on to Maya using the `MFnPlugin::registerCommand` method during the plug-in's `initializePlugin` function (Listing 1).

Loading a plug-in in Maya is done through the Plug-in Manager (Figure 3). This is available from the Windows menu, under Settings > Preferences. The new d3dview.MLL is marked "auto load" so that it will automatically be loaded when Maya is restarted. It is also marked "loaded," indicating that it is currently loaded for use. This ability to dynamically load and unload plug-ins using the "loaded" option can significantly cut turnaround time in developing a plug-in, because Maya just needs to be started once, then the plug-in can be loaded, tested, and unloaded repeatedly.

In Max, the game viewport was created using an extended user viewport. Unfortunately, Maya has no concept of user-created viewports or panels through the SDK (Figure 4). Custom panels and windows can be created through MEL, but these are for holding UI controls, not

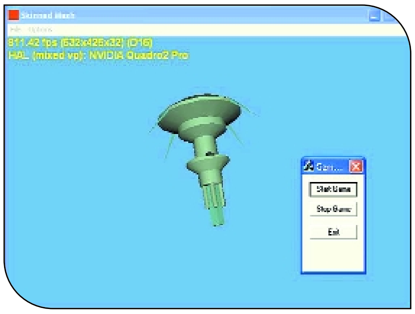
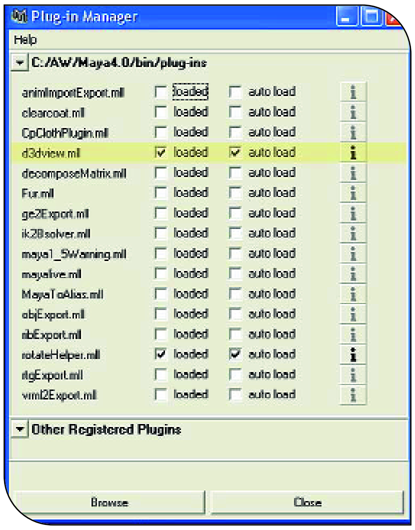

Skinned Mesh
811.42 fps (632x425x32) (D16)
HAL (mixed vp): NVIDIA Quadro2 Pro
Start Game
Stop Game
Exit

FIGURE 2 (above). Direct3D Skinned Mesh sample program running as a DLL. It can be loaded once, then started and shut down repeatedly before exiting.

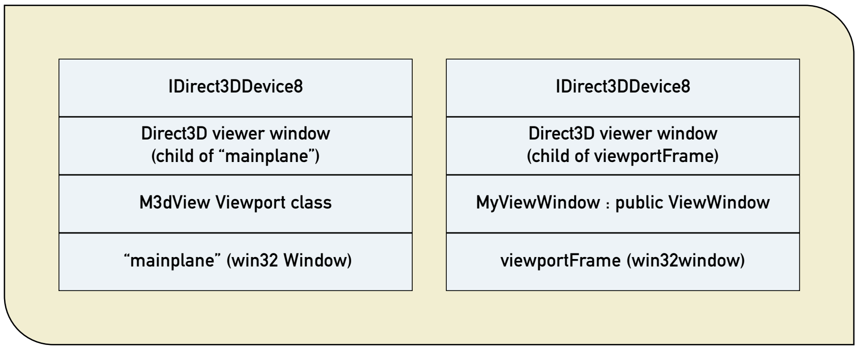
FIGURE 3 (right). The Direct3D viewport viewer is loaded by checking the option in the Plug-In Manager.



C:/AW/Maya4.0/bin/plug-ins

animImportExport.mll	☐ loaded	☐ auto load
clearcoat.mll	☐ loaded	☐ auto load
CpClothPlugin.mll	☐ loaded	☐ auto load
d3dview.mll	☑ loaded	☑ auto load
decomposeMatrix.mll	☐ loaded	☐ auto load
Fur.mll	☐ loaded	☐ auto load
ge2Export.mll	☐ loaded	☐ auto load
ik2Bsolver.mll	☐ loaded	☐ auto load
maya1_5Warning.mll	☐ loaded	☐ auto load
mayalive.mll	☐ loaded	☐ auto load
MayaToAlias.mll	☐ loaded	☐ auto load
objExport.mll	☐ loaded	☐ auto load
ribExport.mll	☐ loaded	☐ auto load
rotateHelper.mll	☑ loaded	☑ auto load
rtgExport.mll	☐ loaded	☐ auto load
vrml2Export.mll	☐ loaded	☐ auto load

Other Registered Plugins



IDirect3DDevice8	IDirect3DDevice8
Direct3D viewer window (child of "mainplane")	Direct3D viewer window (child of viewportFrame)
M3dView Viewport class	MyViewWindow : public ViewWindow
"mainplane" (win32 Window)	viewportFrame (win32window)

FIGURE 4. Ultimately, the Direct3D viewport running in Maya (left table) and Max (right table) look similar. The main difference is that in Max, the developer is actually creating a new type of viewport. In Maya, we rely on Win32 to allow the Direct3D viewport to magically work.

for doing low-level rendering work as is needed here. Therefore, the only recourse is to piggyback off of one of the existing viewports.

With the d3dview.MLL plug-in loaded, the user can switch to the four-pane view, select one of the viewports, and type the new command **d3dviewport** in the MEL command line window. The **d3dviewport** command will create and piggyback the Direct3D viewer window to the selected viewport pane.

Because of the method used to determine which viewport to piggyback on, the user must change to the four-pane view before running the **d3dviewport** command. Even though the static **M3dView::active3dView** method (Listing 3) retrieves the active viewport, this is not always the selected visible pane in the single-pane view. This could be either a bug or a side effect of the way the viewport system is, but you can easily work around it by switching to a multiple-pane mode before running the command.

To view the current scene in the Direct3D viewport, the user need only enter the **gameview** command into the MEL command line. The functionality behind this command is based on the DirectX .X file exporter sample for Maya 4. To simplify managing multiple plug-in files, the sample .X file exporter was merged directly into the project that creates the d3dview.MLL plug-in. The export functionality that was previously exposed through the Maya exporter menu has

LISTING 2. CD3DViewport encapsulates the functionality of the new d3dviewport command. It contains its own static class factory function, CD3DViewport::creator, that Maya uses to instantiate it.

```
class CD3DViewport : public MPxCommand
{
public:
    MStatus        doIt( const MArgList& args );
    static void*   creator();
};

void* CD3DViewport::creator()
{
    return new CD3DViewport;
}
```

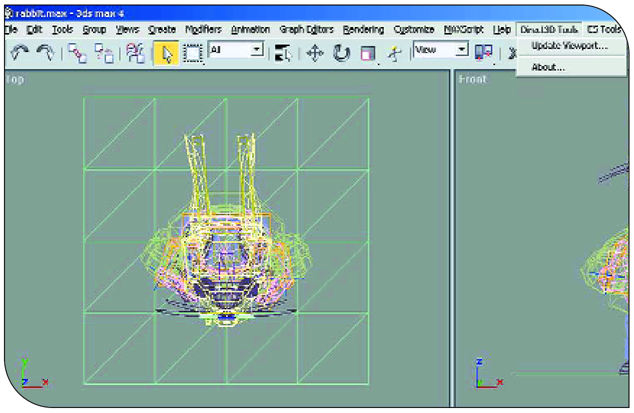


FIGURE 5. A custom menu containing game and export options was added to the 3DS Max main window. Unfortunately, there is no MEL support to do this in Maya.

been placed directly into the gameview command's doIt method (Listing 4).

By executing the gameview command, the current scene is saved off to a temporary file using common .X file export options. The file is then automatically loaded into the Direct3D viewport (Listing 4).

Where Maya Meets the Game

Even though the Direct3D game viewport can be displayed, and the current scene displayed within it, there are still questions to answer about how Maya and the game viewport can interact. The biggest question is when and how the game viewport should be updated from the Maya scene. Although the viewport could be updated in real time by scanning the scene, or when the game viewport receives focus or input, these options are both potentially slow. Also, the user could inadvertently update the game viewport through an errant mouse move.

The best alternative is to have a MEL command, as we do in our example, or preferably a panel window that the user can use to tell the game viewport to transfer the contents of the scene to the game view. While the simple gameview MEL command is enough for transferring the whole scene with some default parameters set behind the scenes, users probably want to use a panel window because they will have various options (selected object? animated?) that they would prefer to select from.

One usability issue that arises from this is how to make the panel window available. In last month's Max implementation of the game viewport, I added a new menu, Direct3D Tools, to the main title bar of the application (Figure 5). This menu contained the option for displaying the panel window where the user could select the options. Creating this sort of menu isn't possible in Maya, because there are no functions available to work directly on the main window menu. An alternative would be to do this using the Win32 API, as I did in the case of Max, but this would break Maya's UI style guidelines because the menu wouldn't be able to be torn off, as can be done with the other menus (excepting the Help menu).

After deciding when to transfer data from the scene into the game viewport, the actual method for transferring the data needs to be selected. This could be done through the file-based scheme used here, through shared memory structures, or the game engine could parse the Maya scene itself. Keeping the data transfer file-based or through memory structures can help keep the Maya-specific code out of the core of the game engine, and in a thin layer on top.

The final integration question is determining how mouse and keyboard input are handled in the game viewport. Again, in contrast to the Max implementation, Maya doesn't have a method for disabling its own processing of keyboard input. This means that the game viewport will need to subclass windows in the window hierarchy and triage input messages, or use an alternative input system such as DirectInput, or make the usage of the game viewport modal. While making the usage of the game viewport modal is a drastic step, it is probably fraught with less trouble than having to use DirectInput and subclass other Maya windows.

But I Use OpenGL!

Even though the example game viewports that were created for Maya and Max are based on Direct3D, an OpenGL-based game engine can be integrated just as easily. In both cases users would create their own OpenGL contexts, instead of Direct3D devices, and handle all of the rest of the window and input handling the same exact way.

Because of its fully integrated support of OpenGL, Maya could make the OpenGL developer's life easier by adding a user-defined viewport. Then the OpenGL developer could use the OpenGL context and functionality exposed through the standard M3dView class.

An alternative for the OpenGL game engine using Maya would be to forgo the

LISTING 3. The MEL `d3dviewport` command determines which viewport is active, obtains its window handle, and passes it on to the function that actually creates the game window and Direct3D device.

```
MStatus CD3DViewport::doIt( const MArgList& args )
{
     RECT rect;
     MStatus status;

     M3dView active3dview = M3dView::active3dView(&status);

     M3dWindow window = active3dview.window(&status);

     HWND hWnd = (HWND) window;

     GetWindowRect(hWnd, &rect);

     createRenderWindow(ghInstance, hWnd, 0, 0,
                        rect.right - rect.left,
                        rect.bottom - rect.top);

    return MS::kSuccess;
}
```

LISTING 4. The MEL `gameview` command exports the current scene to a temporary file and then automatically loads it into the Direct3D viewport for viewing.

```
MStatus CGameView::doIt( const MArgList& args )
{
    char cExportDir[_MAX_PATH];
    GetTempPath(sizeof(cExportDir), cExportDir);

    GetTempFileName(cExportDir, "dvp", 0, gcExportFilename);

    DeleteFile(gcExportFilename);

    // export file
    exportFile(gcExportFilename);

    // load into viewport
    g_d3dApp.loadAndDisplayMesh(gcExportFilename);

    return MS::kSuccess;
}
```

notion of a game viewport and just create one's own Maya objects. This way they could effectively render into all viewports as needed using the built-in OpenGL functionality.

The Good, the Bad

In the end, integrating a Direct3D or OpenGL game engine into Max or Maya can be done. Unfortunately, both packages have similar problems, which will lead to some hard choices for developers looking to create their own in-game viewport.

In the case of Max, the functionality for developing the game viewport exists for the most part. However, there are issues ranging from bugs in handling the input to the custom viewport, to documentation and header files that might lead you to believe that there's more support for integrating Direct3D than there actually is.

The main problem with Maya is that integrating a game viewport is doable, but the developer is going off-road because the SDK isn't set up to support this. There's also no way to create a custom viewport except for UI through a MEL panel, and getting input to the game viewport is problematic because the SDK doesn't have any functionality to tell Maya to stop processing input for a given viewport.

None of the problems encountered with either package is a roadblock to integrating one's own game engine into a viewport, but the developer must plan ahead to reduce the risk of falling into the traps and pitfalls in both packages.

Most of the big differences in developing these plug-ins for Maya and Max come down to the ease with which a plug-in can be developed in Maya, versus the power and flexibility of developing in Max.

Hands down, the process for creating and registering a plug-in for Maya is much simpler than that of Max. Plus there's the added bonus that the plug-in can be dynamically loaded and unloaded while Maya is still running, which can help reduce development time. On the other hand, the Max SDK is much broader and deeper than the Maya SDK, supporting features such as user viewports and the ability to control input processing.

Almost There but Not Quite

As developers continue to bring the art creation pipeline and the game engine closer together, this should be a wake-up call to both Discreet and Alias|Wavefront to address these issues. Alias|Wavefront needs to add the concept of an SDK-created and -managed viewport, and the ability to selectively filter input. If they could extend their unified OpenGL system to support Direct3D, this would be a great help as well, but it's not a necessity. Discreet on the other hand needs to fix their bugs associated with input and viewport handling, and since they already support Direct3D and OpenGL rendering, they should expose it upwards through the viewport system.

The unification of viewports with the underlying rendering system would allow the creation of a completely integrated game engine that could render alongside Max (or Maya) in the viewports. This would create a truly powerful and integrated art pipeline solution.

FOR MORE INFORMATION

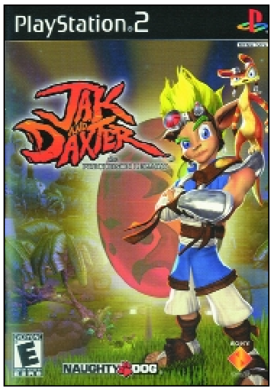
Alias|Wavefront Maya 4
www.aliaswavefront.com

Microsoft Direct3D .X File Exporter for Maya 4
http://msdn.microsoft.com/directx, in the downloads section under DirectX 8.1 SDK Extras

Return

stephen white

Naughty Dog's JAK & DAXTER: THE PRECURSOR LEGACY



GAME DATA

PUBLISHER: Sony Computer Entertainment
NUMBER OF FULL-TIME DEVELOPERS: 35
LENGTH OF DEVELOPMENT: 1 year of initial development, plus 2 years of full production.
RELEASE DATE: December 2001
PLATFORM: Playstation 2
OPERATING SYSTEMS USED: Windows NT, Windows 2000, Linux
DEVELOPMENT SOFTWARE USED: Allegro Common Lisp, Visual C++, GNU C++, Maya, Photoshop, X Emacs, Visual SlickEdit, tcsh, Exceed, CVS

By the end of 1998, Naughty Dog had finished the third game in the extremely successful CRASH BANDICOOT series, and the fourth game, CRASH TEAM RACING, was in development for a 1999 year-end holiday release. And though Sony was closely guarding the details of the eagerly awaited Playstation 2, rumors — and our own speculations — convinced us that the system would have powerful processing and polygonal capabilities, and we knew that we'd have to think on a very grand scale.

Because of the success of our CRASH BANDICOOT games (over 22 million copies sold), there was a strong temptation to follow the same tried-and-true formula of the past: create a linear adventure with individually loaded levels, minimal story, and not much in the way of character development. With more than a little trepidation, we decided instead to say goodbye to the bandicoot and embark on developing an epic adventure we hoped would be worthy of the expectations of the next generation of hardware.

For JAK & DAXTER, one of our earliest desires was to immerse the player in a single, highly detailed world, as opposed to the discrete levels of CRASH BANDICOOT. We still wanted to have the concept of levels, but we wanted them to be seamlessly connected together, with nonobvious boundaries and no load times between them. We wanted highly detailed landscapes, yet we also wanted grand vistas where the player could see great distances, including other surrounding levels. We hoped the player would be able to see a landmark far off in the distance, even in another level, and then travel seamlessly to that landmark.

It was important to us that Jak's world make cohesive sense. An engaging story should tie the game together and allow for character development, but not distract from the action of the game. The world should be populated with highly animated characters that would give Jak tasks to complete, provide hints, reveal story elements, and add humor to the game. We also wanted entertaining puzzles and enemies that would surpass anything that we had done before.

To achieve these and many other difficult tasks required three years of exhausting work, including two years of full production. We encountered more than a few major bumps in the road, and there were times when the project seemed like an insurmountable uphill battle, but we managed to create a game that we are quite proud of, and we learned several important lessons along the way.

What Went Right

1. Scheduling. Perhaps Naughty Dog's most important achievement is making large-scale games and shipping them on time, with at most a small amount of slip. This is an almost unheard of combination in the industry, and although there is a certain amount of luck involved, there are valid reasons to explain how Naughty Dog has managed to achieve this time and again.

Experience will tell you it's impossible to predict the precise details of what will be worked on more than a month or two in advance of doing it, yet many companies fall into the trap of trying to maintain a highly detailed schedule that tries to look too far into the future. What can't be effectively worked into these rigid schedules is time lost to debugging, design changes, over-optimism, illness, meetings, new ideas, and myriad other unpredictable surprises.

At Naughty Dog, we prefer a much more flexible, macro-level

STEPHEN WHITE | *Stephen is the programming director of Naughty Dog, where he has been instrumental in the development of the CRASH BANDICOOT games as well as JAK & DAXTER: THE PRECURSOR LEGACY. Stephen is an industry veteran with over 15 years of published programming experience, including the multi-award-winning Deluxe Paint ST and Brilliance.*

scheduling scheme, with milestone accomplishments to be achieved by certain dates. The schedule only becomes detailed for tasks that will be tackled in the near future. For example, a certain level will be scheduled to have its background modeled by a certain date. If the milestone is missed, then the team makes an analysis as to why the milestone wasn't achieved and changes plans accordingly: the background may be reduced in size, a future task of that artist may be given to another artist to free up more time, the artist may receive guidance on how to model more productively, or some future task may be eliminated.

In the case of JAK & DAXTER, we used the knowledge we'd gained from creating the CRASH BANDICOOT games to help estimate how long it should take to model a level. As we modeled a few levels, however, we soon realized that our original estimates were far too short, and so we took appropriate actions. If we had attempted to maintain a long-term, rigidly detailed schedule, we would have spent a lot of time trying to update something that was highly inaccurate. Beyond this being a waste of time, the constant rescheduling could have had a demoralizing effect on the team.

The game's real-time lighting scheme transitioned through multiple times of day with realistic coloring and shadows.

2. Effective localization techniques. We knew from the start that we were going to sell JAK & DAXTER into many territories around the world, so we knew we would face many localization issues, such as PAL-versus-NTSC, translations, and audio in multiple languages. Careful structuring of our game code and data allowed us to localize to a particular territory by swapping a few data files. This meant we only had to debug one executable and that we had concurrent development of all localized versions of the game.

All of our animation playback code was written so that it could automatically step animations at a rate of 1.2 (60fps/50fps) when playing in PAL. We also used a standardized number of units per second so that we could relate the amount of time elapsed in a game frame to our measure of units per second. Once everything was nice and consistent, then timing-related code no longer had to be concerned with the differences between PAL and NTSC.

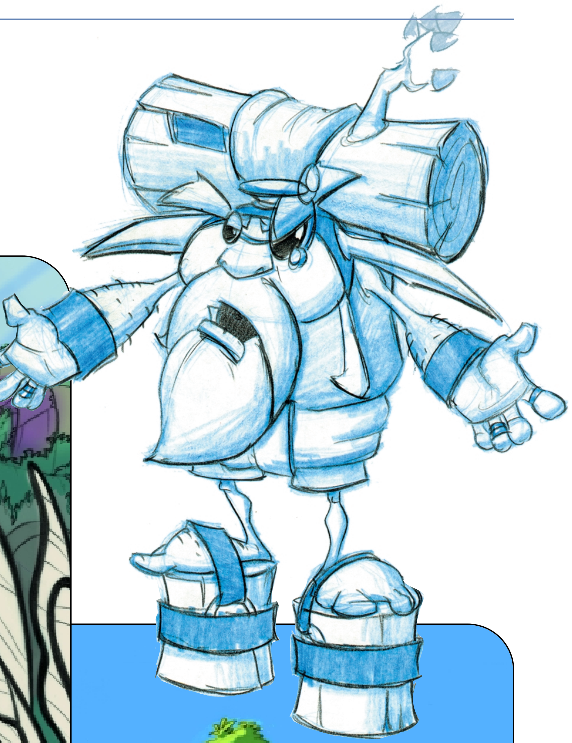
Physics calculations were another issue. If a ball's motion while being dropped is computed by adding a gravitational force to the ball's velocity every frame, then after one second the ball's velocity has been accelerated by gravity 60 times in NTSC but only 50 times in PAL. This discrepancy was big enough to become problematic between the two modes. To correct this problem, we made sure that all of our physics computations were done using seconds, and then we converted the velocity-per-second into velocity-per-game-frame before adding the velocity to the translation.

3. Seamless world, grand vistas, and no load times. We knew very early on in the development of JAK & DAXTER that we wanted to immerse the player within one large expansive world. We didn't want to stall the game with loads between the various areas of that world.

JAK & DAXTER's designers had to overcome many obstacles to achieve our open environments. They had to lay out the levels of the world carefully so that levels could be moved in and out of memory without stalling gameplay or causing ugly visual popping. They also had to create challenges that would engage the player and maintain the player's interest, even though the player could roam freely around the world. And they had to tune the challenges so that the difficulty ramped up appropriately, without giving players the impression that they were being overly directed.

The programmers had to create tools to process interconnected levels containing millions of polygons and create the fast game code that could render the highly detailed world. We developed several complex level-of-detail (LOD) schemes, with different schemes used for different types of things (creatures versus background), and different schemes used at different distances, such as simplified models used to represent faraway backgrounds, and flats used to represent distant geometry. At the heart of our LOD system was our proprietary mesh tessellation/reduction scheme, which we originally developed for CRASH TEAM RACING and radically enhanced for JAK & DAXTER.

The artists had the burden of generating the enormous amount of content for

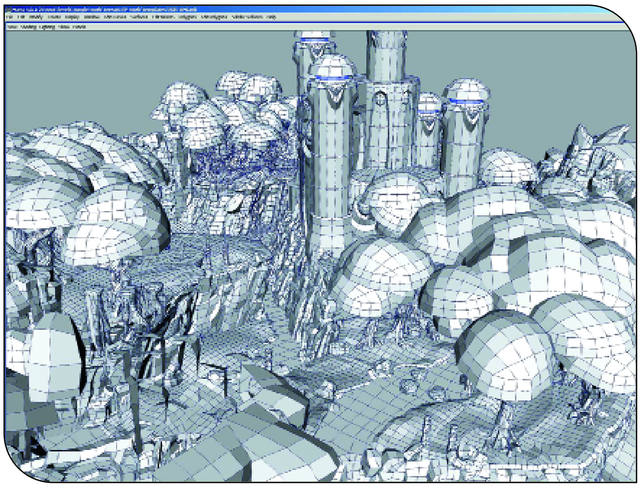


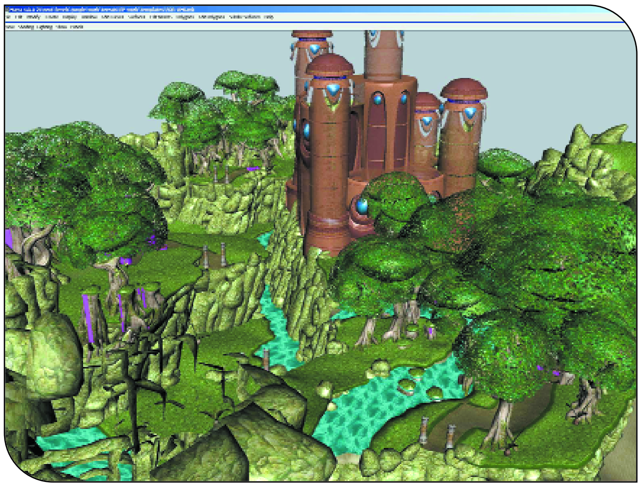
Extensive character sketches and color key/concept design work were done in advance of actual modeling.

these environments. Their task was complicated by the very specialized construction rules they had to follow to support our various renderers. Support tools and plug-ins were created to help the artists, but we relied on the art staff to overcome many difficulties.

4. Camera control. From the initial stages of JAK & DAXTER, we looked at the various camera schemes used in other games and came to the depressing conclusion that all existing camera schemes had serious issues. We suspected that making a well-behaved camera might be an unsolvable 3D problem: How could one possibly create a camera that would maneuver through a complex 3D world while behaving both unobtrusively and intelligently?

Only fools would believe that all problems have a solution, so, like idiots, we decided to give it a try. The resulting camera behaved extremely well, and although it had its limitations, it proved the problem does indeed have a solution. Jak can jump through trees and bushes, duck under archways, run between scaffolding, scale down cliffs, and hide


Screenshot showing highest level of detail of in-game geometry for the Forbidden Jungle level.

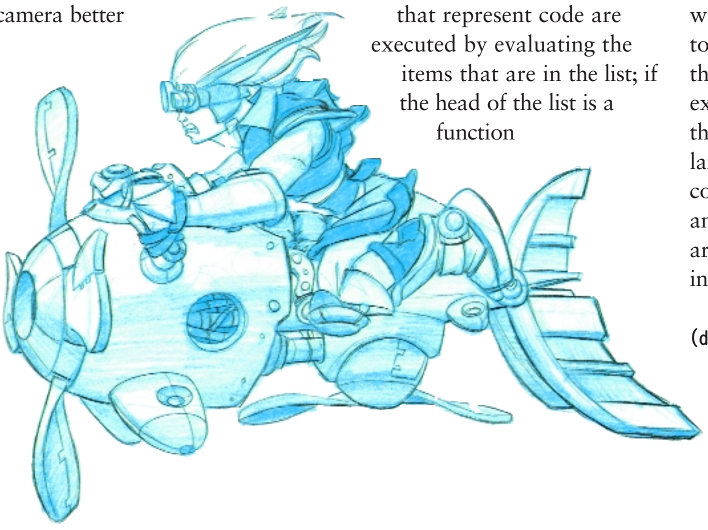

The same shot displayed with textures.

behind rocks, all with the camera unobtrusively keeping the action in view.

We wanted the player to be able to control the camera, but we did not want to force the player to do so. Players can use the second joystick to maneuver the camera(rotating the camera or moving it closer to or farther from Jak), but we were concerned that some people may not want to manipulate the camera, and others, such as children, may not have the required sophistication or coordination. Therefore, we worked very hard at making the camera do a reasonable job of showing players what they needed to see in order to complete the various challenges. We accomplished this through a combination of camera volumes with specially tuned camera parameters and specialized camera modes for difficult situations. Also, creatures could send messages to the camera in order to help the camera better show the action.

This may sound funny, but an important feature of the camera was that it didn't make people sick. This has been a serious problem that has plagued cameras in other games. We spent a bit of time analyzing why people got sick, and we tuned the camera so that it reduced the rotational and extraneous movement that contributed to the problem.

Perhaps the greatest success of the camera is that everyone seems to like it. We consider that a major accomplishment, given the difficulty of the task of creating it.



5. GOAL rules! Practically all of the run-time code (approximately half a million lines of source code) was written in GOAL (Game Object Assembly Lisp), Naughty Dog's own internally developed language, which was based on the Lisp programming language. Before you dismiss us as crazy, consider the many advantages of having a custom compiler.

Lisp has a very consistent, small set of syntactic rules involving the construction and evaluation of lists. Lists that represent code are executed by evaluating the items that are in the list; if the head of the list is a function (or some other action), you could think of the other items in the list as being the parameters to that function. This simplicity of the Lisp syntax makes it trivial to create powerful macros that would be difficult or impossible to implement using C++.

Writing macros, however, is not enough justification for writing a compiler; there were features we felt we couldn't achieve without a custom compiler. GOAL code, for example, can be executed at a listener prompt while the game is running. Not only can numbers be viewed and tweaked, code itself can be compiled and downloaded without interrupting or restarting the game. This allowed the rapid tuning and debugging, since the effects of modifying functions and data structures could be viewed instantaneously.

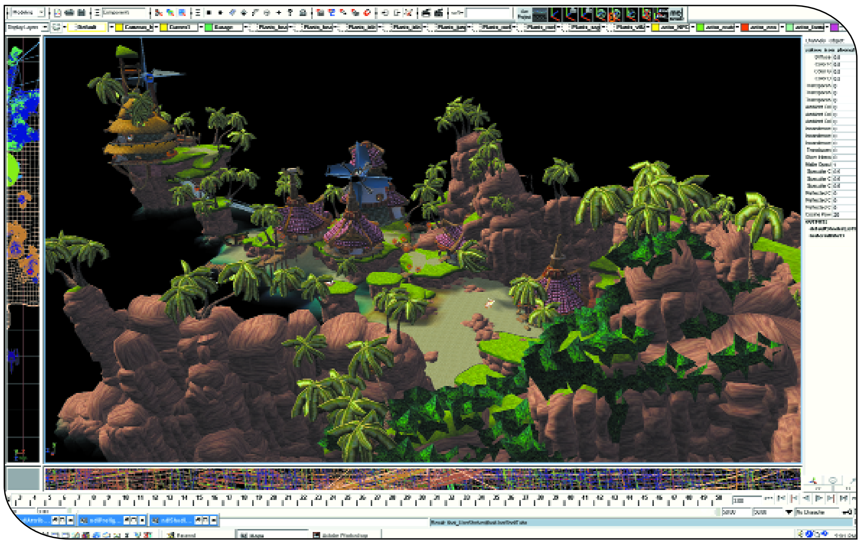
We wanted creatures to use nonpreemptive cooperative multi-tasking, a fancy way of saying that we wanted a creature to be able to execute code for a while, then "suspend" and allow other code to execute. The advantage of implementing the multi-tasking scheme using our own language was that suspend instructions could be inserted within a creature's code, and state could be automatically preserved around the suspend. Consider the following small snippet of GOAL code:

```
(dotimes (ii (num-frames idle))
     (set! frame-num ii)
     (suspend)
        )
```

This code has been simplified to make a point, so pretend that it uses a counter called `ii` to loop over the number of frames in an animation called idle. Each time through the loop the animation frame is set to the value of `ii`, and the code is suspended. Note that the value of `ii` (as well as any other local variables) is automatically preserved across the suspend. In practice, the preceding code would have been encapsulated into a macro such as:

```
(play-anim idle
   ;; Put code executed for each time..
   ;; through the loop here.
   )
```

There are other major compiler advantages: a unified set of assembly op-codes consistent across all five processors of the Playstation 2, register coloring when writing assembly code, and the ability to intermix assembly instructions seamlessly with higher-level code. Outer loops could be written as "slower" higher-level code, while inner loops could be optimized assembly.


Village geometry displayed with textures and sunrise lighting. About 5,000 time-of-day lights were used in the game.

What Went Wrong

1. GOAL sucks! While it's true that GOAL gave us many advantages, GOAL caused us a lot of grief. A single programmer (who could easily be one of the top ten Lisp programmers in the world) wrote GOAL. While he called his Lisp techniques and programming practices "revolutionary," others referred to them as "code encryption," since only he could understand them. Because of this, all of the support, bug fixes, feature enhancements, and optimizations had to come from one person, creating quite a bottleneck. Also, it took over a year to develop the compiler, during which time the other programmers had to make do with missing features, odd quirks, and numerous bugs. Eventually GOAL became much more robust, but even now C++ has some advantages over GOAL, such as destructors, better constructors, and the ease of declaring inline methods.

A major difficulty was that we worked in such isolation from the rest of the world. We gave up third-party development tools such as profilers and debuggers, and we gave up existing libraries, including code previously developed internally. Compared to the thousands of programmers with many years of C++ experience, there are relatively few programmers with Lisp experience, and no programmers (outside of Naughty Dog) with GOAL experience, making hiring more difficult.

GOAL's ability both to execute code at the listener and to replace existing code in the game at run time introduced the problem of memory usage, and more specifically, garbage collection. As new code was compiled, older code (and other memory used by the compiler) was orphaned, eventually causing the PC to run low on free memory. A slow garbage collection process would automatically occur when available memory became sufficiently low, and the compiler would be unresponsive until the process had completed, sometimes taking as long as 15 minutes.

2. Gameplay programming. Because we were so busy creating the technology for our seamless world, we didn't have time to work on gameplay code until fairly late in the project. The situation caused no end of frustration to the designers, who were forced to design levels and creatures without being able to test whether what they were doing was going to be fun and play well. Eventually programmers were moved off of technology tasks and onto gameplay tasks, allowing the designers to play the game and make changes as appropriate. But without our designers' experience, diligence, and forethought, the results could have been a disaster.

3. Audio. We were plagued with audio-related problems from the start. Our first indication that things might not be going quite right was when our sound programmer quit and moved to Australia. Quickly hiring another sound programmer would have been the correct decision. We tried several other schemes, however, made some poor choices, and had quite a bit of bad luck. We didn't recognize until fairly late in development what a monumental task audio was going to be for this project. Not only did JAK & DAXTER contain original music scores, creature and gadget noises, ambient sounds, and animated elements, but there are also over 45 minutes of story sequences, each containing Foley effects and speech recorded in six different languages.

Our audio issues could be broken up into four categories: sound effects, spooled Foley, music, and localized dialogue. Due to the large number of sound effects in the game, implementing sound effects became a maintenance nightmare. No single sound effect was particularly difficult or time-consuming; however, creating all of the sound effects and keeping them all balanced and working was a constant struggle. We needed to have more time dedicated to this problem, and we needed better tool support.

We used spooled Foley for lengthy sound effects, which wouldn't fit well in sound RAM. Spooling the audio had many advantages, but we developed the technology too late in the project and had difficulty using it due to synchronization issues.

Our music, although expertly composed, lacked the direction and attention to detail that we had achieved with the CRASH BANDICOOT games. In previous games, we had a person who was responsible for the direction of the music. Unfortunately, no one performed that same role during JAK & DAXTER.

Dialogue is a difficult problem in general due to the complexity of writing, recording, editing, creating Foley, and managing all of the audio files, but our localization issues made it especially challenging. Many problems were difficult to discover because of our lack of knowledge of the various languages, and we should have had more redundant testing of the audio files by people who were fluent in the specific languages.

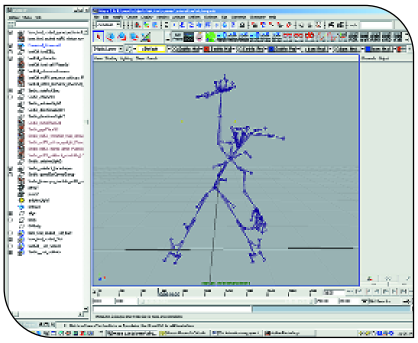
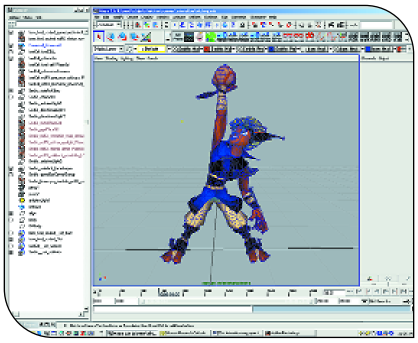
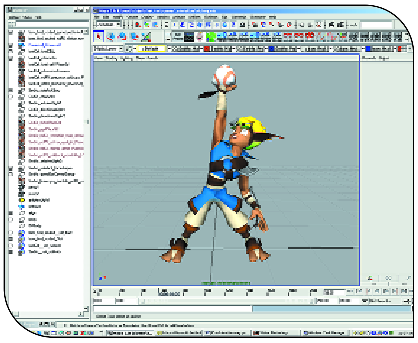
Color key for Forbidden Jungle. The game required two full-time conceptual artists for nearly two years of production.

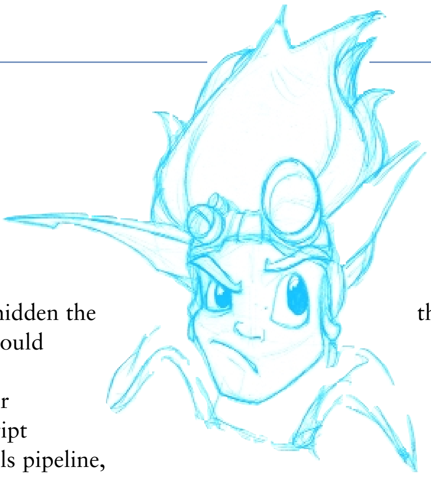
4. Lengthy processing times.

One of our greatest frustrations and loss of productivity came from our slow turnaround time in making a change to a level or animation and seeing that change in the actual game.

Since all of our tools (including the GOAL compiler) ran across our network, we ran into severe network bandwidth issues. Making better use of local hard drives would have been a smarter approach. In addition, we found extreme network slowdown issues related to reading file time/date stamps, and some tools took several minutes just to determine that nothing needed to be rebuilt. When we compiled some of our tools under Linux, we noticed dramatic improvements in network performance, and we are planning on using Linux more extensively in our next project.

We implemented the processing of the lengthy story-sequence animations as a hack of the system used to process the far simpler creature animations. Unfortunately, this bad system caused lengthy processing times, time-consuming debug-




A frame from Jak's victory animation displaced as IK, textured, and textured with color shading.



ging, and a lot of confusion. If we had initially hidden the processing complexity behind better tools, we would have saved quite a bit of time.

We used level-configuration scripts to set actor parameters and other level-specific data. The script processing was done at an early stage in our tools pipeline, however, so minor data changes took several minutes to process. We learned that tunable data should instead be processed as close as possible to the end of the tools pipeline.

5. Artist tools. We created many tools while developing JAK & DAXTER, but many of our tools were difficult to use, and many tools were needed but never written. We often didn't know exactly what we needed until after several revisions of our technology. In addition, we didn't spend a lot of time polishing our tools, since that time would have been wasted if the underlying technology changed. Regrettably, we did not have time to program tools that were badly needed by the artists, which resulted in a difficult and confusing environment for the artists and caused many productivity issues. Since programming created a bottleneck during game production, the added burden given to the artists was considered necessary, though no less distasteful.

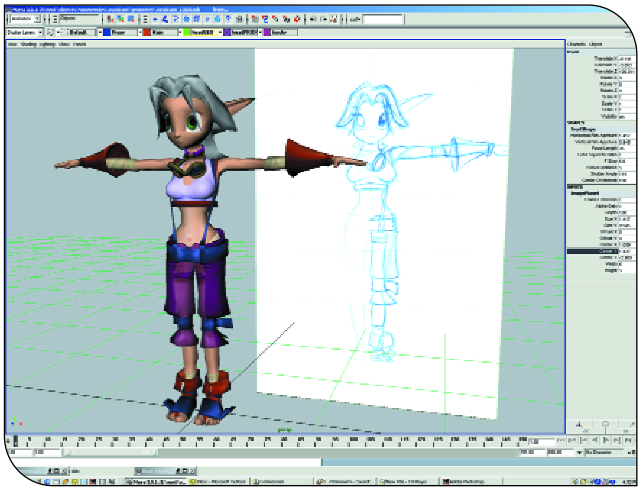
We lacked many visualization tools that would have greatly improved the artists' ability to find and fix problems. For example, the main method artists used to examine collision was a debugging mode that colorized a small section of collision geometry immediately surrounding Jak. A far better solution would have been to create a renderer to display the entire collision of a level.

We created plug-ins that were used within the 3D modeling package; however, for flexibility's sake most of the plug-ins operated by taking command parameters and outputting results as text: not a good interface for artists. Eventually, one of our multi-talented artists created menus and other visualization aids that significantly improved productivity.

Many of our tools were script based, which made the tools extremely flexible and adaptable; however, the scripts were often difficult for the artists to understand and use. We are replacing many of these scripts with easier-to-use GUIs for our next project.

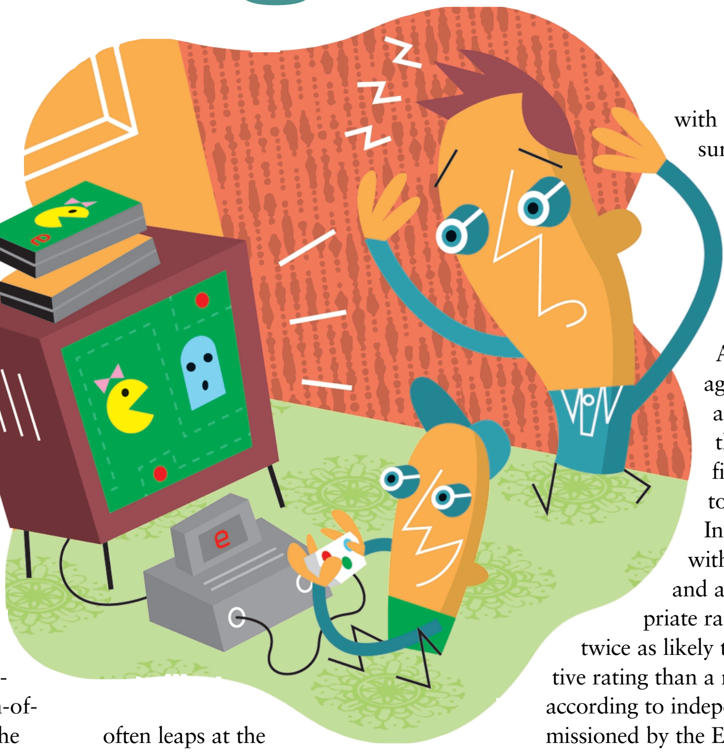
The Legacy

Creating JAK & DAXTER was a monumental effort by many hardworking, talented people. In retrospect, we were probably pretty foolish to take on as many challenges as we did, and we learned some painful lessons along the way. But we also did many things right, and we successfully achieved our main goals. At Naughty Dog, there is a strong devotion to quality, which at times can border on the chaotic, but we try to learn from both our success and our failures in order to improve our processes and create a better game. The things that we learned from JAK & DAXTER have made us a stronger company, and we look forward to learning from our past as we prepare for the new challenges ahead.


Front and side view drawings of the character are scanned, imported, and used as blueprints for modeling.

Return

Staying Ahead in the Ratings Game



Since its creation in 1994, the Entertainment Software Rating Board (ESRB) has assigned ratings and content descriptors to more than 9,000 computer and videogames to help parents and consumers choose games that are right for them and their families.

When the ESRB was created, advocacy groups and some government leaders were highly critical of the interactive software industry — a few had even called for the imposition of a government regulatory system to monitor game content, despite serious First Amendment and freedom-of-commerce concerns. Fortunately, the ESRB's rating system and our aggressive efforts at public outreach have won over many of these critics.

Unlike other entertainment industry rating systems, the ESRB system is truly content based. The rating icons (EC, E, T, M, and AO) provide an at-a-glance reference for age appropriateness, but the content descriptors alert consumers to game elements they may wish to search out — such as educational content — or elements they may wish to avoid, such as violent or sexual content.

More than anything else, the ESRB's emphasis on content has earned the praise of many former critics, including Senator Joseph Lieberman (D-Conn.), who has described the ESRB system as "the best entertainment rating system in America."

Despite the ESRB's success, some industry critics continue to attack the rating system. And when they do, the news media often leaps at the story, often without carefully examining the critics' claims.

While the ESRB welcomes consumer comments and well-intended suggestions, we will not change the rating system in response to the kind of incomplete or misleading criticism that I will focus on here. Let's take a closer look at two specific charges leveled at the ESRB over the past year.

Criticism #1: The ratings are "soft" or too lenient. Some critics maintain that more games should be rated T or M, and they imply that the ESRB is "soft" on game publishers because (in their view) too many games receive an E rating. (Publishers who have received ratings more restrictive than they had hoped know this criticism is truly off-base.)

These critics generally rely on studies showing that people sometimes disagree with ESRB ratings, citing surveys showing that some people would assign a T rating instead of an E to certain games. What these critics fail to point out is that Americans overwhelming agree with ESRB ratings, and when they disagree, they are more likely to find the ESRB's rating too strict than too lenient. In fact, when presented with actual game footage and asked to select the appropriate rating, consumers are twice as likely to choose a less restrictive rating than a more restrictive one, according to independent research commissioned by the ESRB.

These results are neither surprising nor indicative of problems with the ESRB system. In a country as diverse as ours, it's impossible to achieve 100 percent agreement on anything. For example, some parents believe that games depicting outdoor activities such as hunting or fishing are appropriate for young children, whereas others choose to shield their families from such activities.

What all this research really shows is that ratings are subjective — that well-intentioned people can examine the same game and, when it's a close call, may reach different conclusions about the most appropriate rating. Accepting this reality, the ESRB never intended to design a rating system that would foster 100 percent agreement. Rather, our goal is to administer a system that is helpful to parents and reflects the values of the

continued on page 71

Illustration by Darren Raven

continued from page 72

vast majority of parents and consumers.

Criticism #2: E-rated games contain excessive violence. One recent study printed in the *Journal of the American Medical Association* went so far as to charge that nearly two-thirds of E-rated games contained acts of violence. The central problem with this study, and others like it, is its unusually broad definition of violence. These critics discern violence in innocent acts that most Americans would dismiss. For example, the *JAMA* report cited MS. PAC-MAN as a violent game, because players receive points for eating ghosts. In a similar study of children's films, critics identified acts of violence in every G-rated animated film ever made, including *Dumbo* (because Dumbo shoots peanuts through his trunk during a circus performance).

Just as it is impossible to design a rating system that produces 100 percent agreement, it's equally impossible to define violence in such a way as to produce universal approval (although defining PAC-MAN and CENTIPEDE as violent games is clearly not the solution). For example, some parents may find the physical contact associated with sports like hockey to be inappropriate for young children, while others may consider such sports to be good, clean fun.

The ESRB respects both points of view. It is not our role to tell families which games are appropriate for kids. That's a judgment families need to make for themselves. Instead, we strive to give parents and consumers the information they need to make informed choices. And according to both independent research and the government leaders who watch our system closely, we're achieving that objective every day.

ARTHUR POBER, ED.D. | *Dr. Pober is president of the Entertainment Software Rating Board.*