GAME DEVELOPER MAGAZINE

APRIL 2001

UBM Tech

Return

GAME PLAN

LETTER FROM THE EDITOR

Whole New Ball *Game*

Welcome to *Game Developer* magazine's new look! As I've mentioned in previous issues, we have been making some changes around here based on your feedback. Nothing drastic, just some little tweaks around the edges to provide you with more information that you can use in your day-to-day game development.

At the bottom right of this page you'll find our group's new name: Gama Network. We were previously known as the CMP Game Media Group. Why did we change our name? We want you to know that we're the same group that runs the Game Developers Conference and Gamasutra.com. We've changed all of our logos to have a consistent look. We hope you like the new style of our front cover as much as we do!

But we wouldn't be *Game Developer* without great articles. This month we introduce our first developer profile. Each month we'll interview a developer you should know, someone who has a unique industry perspective. For our first profile we introduce you to Super Duper Game Guy Ed Logg (seriously, that's his title). Ed is perhaps the hardest- and longest-working game programmer out there. His credits include the arcade titles ASTEROIDS, CENTIPEDE, GAUNTLET, and TETRIS; and more recently, the console versions of SAN FRANCISCO RUSH, RUSH 2, and RUSH 2049.

If you turn to page 10, you'll find our new and expanded Product Review section. We've received consistent feedback that one of the most valuable parts of the magazine for you are the product reviews. No other magazine covers products from a game development perspective, so we've increased our monthly coverage from one review each month to four. We're also going to increase the scope of our reviews to include software, hardware, books, even game engines and SDKs. The number of tools for game developers to use out there just keeps increasing, and we want to be your first resource when you're looking for new tools.

This month our Postmortem is on AMERICAN MCGEE'S ALICE, by Rogue Entertainment. Alice uses the QUAKE 3 engine to great advantage. The first time I saw a room split itself in half I was pretty impressed. Read about what trials and tribulations the Rogue staff went through as they took Lewis Carroll's crazy tale of *Alice in Wonderland* and made it even crazier! (Please excuse the blatant Old Man Murray reference.)

Our main feature this month is on a similarly crazy game, MAJESTIC. (You could say this is our crazy issue. Or our EA issue. Or our Ed Logg issue — Ed is also mentioned in our Soapbox). Imagine that you're playing a PC game, but someone keeps calling you on your cell phone and screaming at you incoherently about the developers that worked on the title. It seems they're in some kind of trouble. You hop on the Internet to find out more and follow a link on Google to a newspaper article describing a fire at the company's headquarters. Suddenly someone sends you an instant message and hints that these developers have gone underground and you should check a particular voice-mail box to find out more information. Is this a game? Who are these developers, anyway? MAJESTIC is an episodic game that plays on the boundary between fantasy and reality. Read our feature on what it took to create MAJESTIC, both from a design perspective and a technological standpoint.

Our second feature this month comes from an analysis of the troubles with using the A* pathfinding algorithm. Marco Pinter's article details ways that you can smooth out the jagged paths generated by A*. Then he extends A* to include an initial and destination orientation. Check out Marco's modifications, which he terms the Directional A* algorithm.

We hope you enjoy this month's issue. Let us know what you think about our brand new look!

[illegible]

GameDeveloper

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

Publisher
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL

Editor-In-Chief
Mark DeLoura *mdeloura@cmp.com*

Senior Editor
Jennifer Olsen *jolsen@cmp.com*

Managing Editor
Laura Huber *lhuber@cmp.com*

Production Editor
R.D.T. Byrd *tbyrd@cmp.com*

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

Contributing Editors
Daniel Huebner *dan@gamasutra.com*
Jeff Lander *jeffl@darwin3d.com*
Mark Peasley *mpeasley@gaspowered.com*

Advisory Board
Hal Barwood LucasArts
Noah Falstein The Inspiracy
Brian Hook Independent
Susan Lee-Merrow Lucas Learning
Mark Miller Group Process Consulting
Paul Steed WildTangent
Dan Teven Teven Consulting
Rob Wyatt The Groove Alliance

ADVERTISING SALES

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

National Sales Manager & Western Region, Silicon Valley & Asia
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, Recruitment
Morgan Browning e: *mbrowning@cmp.com* t: 415.947.6225

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

Sales Associate
Aaron Murawski e: *amurawski@cmp.com* t: 415.947.6227

ADVERTISING PRODUCTION

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

CMP GAME MEDIA GROUP MARKETING

Senior MarCom Manager Jennifer McLean
Strategic Marketing Manager Darrielle Ruff
Marketing Coordinator Scott Lyon
Audience Development Coordinator Jessica Shultz
Sales Marketing Associate Jennifer Cereghetti

BPA INTERNATIONAL
Game Developer magazine is BPA approved

CIRCULATION

Group Circulation Director Kathy Henry
Director of Audience Development Henry Fung
Circulation Manager Ron Escobar
Circulation Assistant Ian Hay
Newsstand Analyst Pam Santoro

SUBSCRIPTION SERVICES

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

INTERNATIONAL LICENSING INFORMATION

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

CMP MEDIA MANAGEMENT

President & CEO Gary Marshall
Corporate President/COO John Russell
CFO John Day
Group President, Business Technology Group Adam K. Marder
Group President, Specialized Technologies Group Regina Starr Ridley
Group President, Channel Group Pam Watkins
Group President, Electronics Group Steve Weitzner
Senior Vice President, Human Resources Leah Landro
Senior Vice President, Global Sales & Marketing Bill Howard
Senior Vice President, Business Development Vittoria Borazio
General Counsel Sandra Grayson
Vice President, Creative Technologies Johanna Kleppe

GamaNetwork
A DIVISION OF CMP MEDIA INC.

WWW.GAMANETWORK.COM

Return

SAYS YOU

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

Fat Men Tell No Lies

I just wanted to drop you a quick note and compliment you on your recent music article in *Game Developer* ["The Sound of Money (Down the Potty): Common Audio Mistakes in Kids' Games," February 2001]. It was one of the best I have read in a while! You made me think of new sound issues, and you even made it sound interesting. I hope The Fat Man can write more articles in the future, perhaps on specific sound/music issues (for example, creating long, nontedious background music). Thanks!

Steve Sheets
Midnight Mage Software
via e-mail

I enjoyed The Fat Man's article in the February issue of *Game Developer*. After reading it, I decided to consider myself lucky that for several years my biggest client has been Disney Interactive. Overlooking for the moment the fact that I always think I should be paid more, at least I haven't had to deal with most of the audio mistakes listed in your article — probably because Disney really is an entertainment company, and the story really is more important to them than the computer code. I loved the article's anecdote about letting kids save their games. How many times have I said to my own children, "It's time for dinner," and the reply is always, "Just a minute, as soon as we finish this level!" Thanks for sharing.

Billy Martin
Lunch With Picasso Music
via e-mail

Put Effort Where It's Due

In response to Mark DeLoura's February editorial ("Telling Stories," Game Plan), I agree that story is important for certain kinds of games, but I don't think it's as all-important as you make it out to be.

Games are about interactivity. What we need to do is not to mimic the sort of author-predetermined linear story that we see in books and film, but to come up with our own kind of nonlinear, interactive story that is appropriate to our medium. Games are never going to be as good at telling a linear story as books and film (because that involves the supression of the player's free will, and games are about expression of free will), so we should not try to compete with them.

The reason we don't make this kind of nonlinear, interactive story is that we don't know how. Our best attempt at it so far was DEUS EX . . . which did a pretty good job, all things considered, by just throwing tons of content at the problem. But it's obvious that this approach will not scale indefinitely into the future.

What we need to do is invent a technological system that allows us to create things that are better at interacting with people in rich and meaningful ways. You make the point that graphics are so developed that players couldn't see the difference with future graphical improvements. I think that is largely true (though only because today's games are so primitive in other ways). I think that we need to focus a lot more technological development in games but not on graphics.

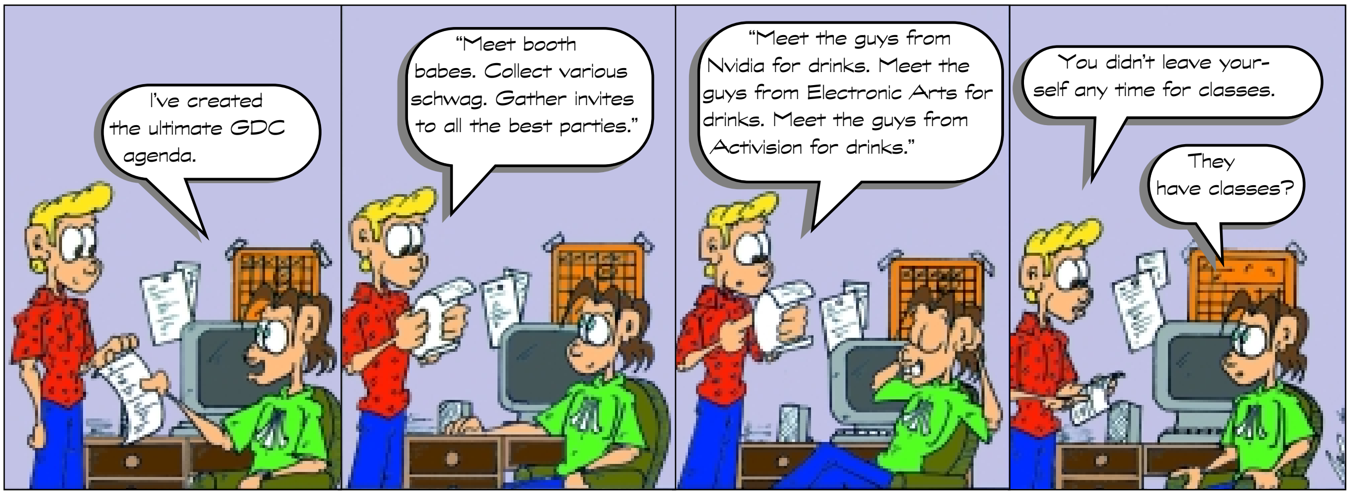
Compared to our current graphics technology, our input technology sucks. Controls are almost always a limiting factor in games. Our AI sucks. We can't make a person in a game that you can talk to, or that can talk back. We can't even dynamically synthesize the sound of a fork scraping across the table, much less the sound of a person speaking emotionally.

Games need to become something very different from what they are now. But they won't be able to become whatever it is they need to be until we do enough technological development to open the right doors.

Jonathan Blow
Bolt Action Software
via e-mail

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

Kludge by Tiger Byrd and Daniel Huebner


I've created the ultimate GDC agenda.
"Meet booth babes. Collect various schwag. Gather invites to all the best parties."
"Meet the guys from Nvidia for drinks. Meet the guys from Electronic Arts for drinks. Meet the guys from Activision for drinks."
You didn't leave yourself any time for classes.
They have classes?

Return

INDUSTRY WATCH

daniel huebner | THE BUZZ ABOUT THE GAME BIZ

SONIC THE HEDGEHOG. Sega's trademark character will soon appear on platforms of his fomer rivals.

Sega pulls the plug on Dreamcast. Sega's plans to stem the tide of red ink in its console business have finally become clear. After weeks of speculation, the company confirmed that it would cease production of its Dreamcast console on March 31, the end of its fiscal year. Dreamcast sales have been disappointing, falling 44 percent below Sega's projection — though the console shutdown will be costly as well. Sega anticipates that killing the Dreamcast will cost $689 million, pushing Sega's loss for the fiscal year to a record $500 million, as well as necessitating workforce reductions.

Though Sega has pledged to continue to offer support for the Dreamcast with more than 100 titles planned for this year, the company has also confirmed that Sega titles will appear on rival platforms. Sega has already locked up deals to bring its games to Sony's Playstation 2 and Nintendo's Game Boy Advance, with VIRTUA FIGHTER 4 tabbed for the PS2 and SONIC THE HEDGEHOG heading to GBA. Sega is working on arrangements to extend its brand to include Xbox and Gamecube titles.

One additional avenue for Dreamcast's post-Sega life is as a set-top box. Sega and Pace Micro Technology have reached an agreement that will add Sega's Dreamcast console technology to a set-top box. Pace will pair the game platform with its digital personal video recorder system, a set-top box with an integrated hard drive that is compatible with multiple cable and satellite systems, allowing on-demand access to Sega's videogame titles.

PS2 problems cut Sony profits. Sony's Playstation 2 problems led to a big hit to the company's bottom line. Sony's third quarter, which runs from October 1 to December 31 and includes the critical holiday season, showed an 11 percent drop in profits over the same quarter last year. Sony is placing the blame for most of that decline in profits squarely on PS2 production delays caused by parts shortages, though the slowing pace of the U.S. economy was also problematic. Profits in Sony's game division were down a whopping 23 percent from the same period last year. All together, Sony still managed a group operating profit of $1.24 billion for the quarter. The company's rough patch is still far from over, as Sony has been forced to cut its profit projections for the fiscal year in half and Playstation 2 shipments by 10 percent.

Sony acquires two key developers. Though a solution for its Playstation 2 production woes seems to be just beyond Sony's reach, the company is taking steps to take tighter hold of the development of some of its top franchises. CRASH BANDICOOT has long been closely associated with Playstation; now Sony Computer Entertainment America (SCEA) is solidifying that relationship by acquiring CRASH creator Naughty Dog. Naughty Dog and its entire development team will become a wholly-owned subsidiary of SCEA and will develop exclusively for Playstation 2. The company will keep the Naughty Dog name and continue on under the direction of its founders, Andy Gavin and Jason Rubin.

Sony is also bringing its sports game development in-house by acquiring Red Zone Interactive, the developers of the NFL GAME DAY series published under Sony's 989 Studios label. Unlike Naughty Dog, Red Zone will become a division of SCEA's first-party development operations under the direction of Shuhei Yoshida. Red Zone's 65 employees will continue to work out of Red Zone's San Diego office exclusively on Playstation titles. Sony did not release the financial terms of either deal.

The Learning Company renames entertainment division. Game maker The Learning Company is hoping that a new name will exorcise its demons. Now called Game Studios, the Novato, Calif.–based developer will gather The Learning Company, Broderbund, Mindscape, SSI, and Red Orb under the new Game Studios banner, although the Strategic Simulation Inc. imprint will continue on the company's strategy and simulation titles. Game Studios will continue on as both a third-party publisher and as a developer of PC and console-based titles.

Activision exceeds estimates. Activision's third-quarter profits managed to beat analyst predictions by 10 cents per share, with the company reporting income of $20.5 million on revenues of $264.5 million — slightly down from last year's third-quarter result of $22.3 million in income on $268.9 million in revenue. The encouraging results have led Activision to increase its full-year earning predictions by nine percent to $48 million.

UPCOMING EVENTS CALENDAR

ELECTRONIC ENTERTAINMENT EXPO

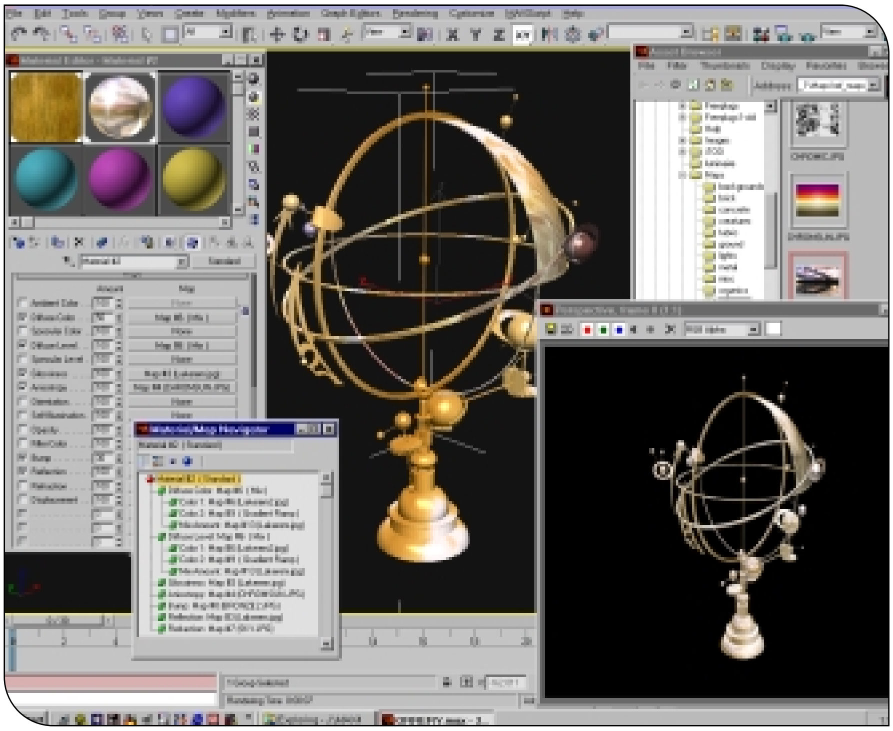
LOS ANGELES CONVENTION CENTER
Los Angeles, Calif.
Conference: May 16–18, 2001
Expo: May 17–19, 2001
Cost: $200–$450
www.e3expo.com

Return

Discreet's 3DS Max 4

by jeff abouaf

Atop the users' wish list last year for Discreet's 3D Studio Max was a redesigned bones and forward/inverse kinematics chain system (FK/IK) for animating hierarchical characters. Maybe it was unfair to ask for this, given that release 3 contained serious modeling improvements and a rework of shaders, materials, lights, shadow, antialiasing, and rendering, not to mention interfacing with Mental Ray and Renderman, and a direct link to Discreet's new postproduction products, Paint and Effect (now combined into Combustion). Character animators were disappointed, and competitors disparaged the product. Discreet representatives told me last year that there were but a handful of persons qualified to rework Max's hierarchical animation code. Discreet listened to their users and hired the talent for this release.


Max 4 addresses complaints about 3DS Max's interface with improved mouse utility.

New Max Bones and FK/IK Features

Max 4 sports a major rewrite of forward/inverse kinematics, bones, solvers, and constraints. Character Studio 3 (CS3) expanded Biped's foot animation with an animatable pivot point, and with new flocking, behaviors, and crowd control. Physique was rewritten for speed and ease of use. These simultaneous efforts to upgrade CS3 and Max's chain-based character animation represent a major focus of resources and commitment by Discreet to expand their position in the broadcast, game, and emerging online interactive markets.

Max bones now no longer flip when constrained. Under a new animation menu, you control bone hierarchies through a series of IK solvers and constraints. With the new FK/IK arrangement, dummy objects can be used like handles to drive a bones system properly set up. One dummy handle could drive a character's hip, knee, and ankle joints, while another might move and bend the foot. An animator could drive a very natural walk cycle using 12 bones (six per leg) controlled by four handles (two per leg). Also, bones can be now rescaled in height and width, and include optional "fins" that can protrude top, bottom, and sides. Bones also can be shaded for volume display and previews. The Skin and Physique modifiers respond to the scale and fins, enabling a better skeletal fit to the character mesh. The envelopes and tendons in the Physique modifier (and envelopes in Skin) are no longer the only way to tune bone-driven mesh deformations: lattice constraints can be linked to changes in bone angles or positions to portray muscle flexing, or they can be applied directly to a joint and animated. There are three types of deformers: joint angle deformer, bulge angle deformer, and morph angle deformer. The Skin modifier features include painting weight attributes on the surface.

Animators used to Maya or Softimage should feel more at home with the new Max IK for the greater control, ease of use, and mainstream character setup and animation conventions. My wish list for the next release is to see more AI-based constraints built into Max bones and skeletons. This can save both setup and animating time. I'd also like the new FK/IK bones to support creating, saving out, and blending nonlinear animation components.

Modeling Enhancements

The modeling tools continue their evolution toward more control and ease of use. For patch modelers and Surface Tools users, spline-cage modeling tools offer easier access Bézier handles when splines intersect in 3D space. Keyboard shortcuts are available for Patch modeling, just as they were for editable meshes in Max 3.1, and are now available for every command. The Edit Patch modifier includes soft-selections.

The Flex modifier, used to portray secondary animation and soft-body dynamics, is expanded for use in cloth simulation. Meshsmooth's subdivision surface capabilities are enhanced to include soft-selections and expanded reset features. Max 4 includes Intel's Multi-Resolution Mesh technology (MRM), as well as Hierarchical

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

Subdivision Surfaces (HSDS). MRM permits continuous and selective polygon decimation, useful to solve level-of-detail issues in real-time 3D. With these enhancements, Max can res-up or simplify polygonal models as never before. A new polygon modeling tool converts triangular faces into quads, providing advantages in both modeling and UVW mapping.

UI Enhancements

The customized user interface (CUI) options have been expanded and reorganized in Max 4. You have a very large CUI dialog encompassing the keyboard, toolbars, quads, menus, and colors. You have options to record macros and turn them into buttons on either tabs or toolbars. As before, you can load these on the fly, only now you have greater control over features, which let you redesign and simplify this massive toolset to meet your needs.

Nested menus on the edge of a screen force you to look away from your work to dig and find a tool. In Max 1, nested menus were replaced by buttons nesting rollouts nesting more buttons, and so on. New users complained that there were too many buttons. Discreet has taken two steps to fix this: maximizing right-click access with new context-sensitive (and customizable) quad menus, and opening up the Command Panel. In Max 3, you could reposition, float, and hide the Command Panel. Alternatively, you could use the Tab Panel to get to most features nested in the Command Panel. Now, instead of rollouts going down to the floor, you can expand the Command Panel across the UI by dragging its left edge toward the right. This lets you get to top and subfeatures at once. Also, the Modifier Panel is an expandable outline (no longer a drop-down); you can reorder the stack by drag and drop. In addition to changing viewport layouts, you can also resize any viewport by dragging its border right or left, up or down.

The new quad menus further maximize mouse utility. A right click brings up a square icon divided into four parts, each part giving you a separate menu. While it doesn't do away with nested menus, you can access four top levels immediately. It also remembers the last several tools used, so if you are doing a series of extrude-scale operations, these features remain at the top of one of the quads. A three-button mouse would use the left button for selection and manipulation, the center for navigation (with the Alt and Control keys), and the right for context-sensitive tool selection. The quad menus can be customized and saved with different CUIs, and, if desired, you can view a sound file in the main UI.

Texturing Enhancements

Multi-map materials can now be displayed in the viewport. Instead of a single map, you can examine the blending or compositing of any map tree in the viewport. You can also see vertex illumination and alpha channel effects in the viewports. If you are using DirectX 8, you can display vertex and pixel shaders like bump maps and reflection maps. With the new Active Shade renderer, you have an interactive, continuously updating final render window of all or part of the scene. These features speed resolving material, mapping, and lighting issues in the scene. The ability to drag and drop materials and maps from the Material Navigator, Map Browser, and map channels remains unchanged.

Max 4 contains improved mapping for patch models, as well as improved depth of field and motion blur render effects. The Asset Manager utility, renamed Asset Browser, is expanded to search across networks and online for texture maps and geometry files.

Miscellaneous Features

Discreet designed and will support Max 4 on Windows 2000 only; this means support for HEIDI, OpenGL, and Direct3D drivers. They stated early in the beta program they would not support Max 4 on NT 4. But my Max 4 experience on NT 4 was about the same as for Max 3.1.

Max 4 no longer requires a hardware lock. The new C-dilla software security system generates an authoring code based on the computer hard disk, and not from an external lock. You can no longer easily move your Max installation from one machine to another. I argued against this for release 3 and am disappointed to see it now. Tying software to a single computer restricts the mobility of an independent Max artist; with a hardware lock, you could work on-site for a client who didn't own Max by taking your lock, the installation disk, and a Zip disk containing a personal keyboard, CUIs, plug-ins, and texture maps. With the new software lock, you must either transport your computer(s) to the site, or contact Discreet to authorize your client's hardware as part of a pool. Either result is cumbersome and wasteful.

Final Word

Discreet deserves praise for fitting Max 4 with improved character animation tools. It includes a very adequate animation alternative for those disliking CS3. Last summer, Discreet announced a reorganized focus "on film, on air, online." The Max 4 release contains added value for each of these areas, and should be well received.

3DS MAX 4 ★★★★½

STATS

DISCREET
Montreal, Quebec
(514) 393-1616
www.discreet.com

PRICE
$3,495

SYSTEM REQUIREMENTS
Windows 98/2000 (SP1) with 300MHz Intel-compatible processor(s), 128MB RAM (256MB recommended), and 400MB hard disk space (300MB swap); CD-ROM; sound card and speakers; pointing device; optional network card.

PROS
1. Bones and IK system rewritten and updated. If used with Character Studio 3, this represents a major step forward in character animation capability.
2. Enhancements to modeling tools for game modeling.
3. Builds on strengths of prior releases and continues evolutionary enhancements.

CONS
1. Software security system may impede artist's ability to travel with the software.
2. New features and capabilities bring more complexity and a new learning curve.
3. Max 4 requires experience and long-term commitment.

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

IK MULTIMEDIA'S T-RACKS 24 2.0

by aaron marks

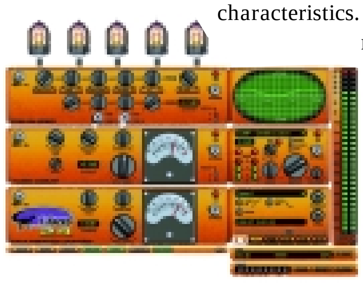
Game musicians and sound designers have yet another software application to add to their bag of tricks — T-Racks 24 2.0. Available for both Windows and Macintosh at a very reasonable price of $299, this new and improved edition doesn't generate unusual noises, add dazzling effects, or inspire wild creativity; it simply takes what you've already created and makes it better.

"Mastering" is not usually a process synonymous with game audio. Audio production is typically fragmented with delivery of music cues and sound effects spread over several milestone submissions. With larger games, it is quite possible to have a wide range of volume and equalization mismatches that are painfully noticeable. A final mastering session sees to their overall uniformity and ensures consistent playback quality, giving the final mixed audio a certain characteristic which ties it to the game. T-Racks 24 will refine any audio production when used in conjunction with patience and a good set of ears.

In this age of cold, almost brittle digital recording methods, T-Racks 24 adds a pleasant analog warmth, clarity and presence, and either subtle or extreme equalization and volume adjustments through a simple, single screen interface.

Full 24-bit resolution enables up to 24-bit .AIFF, .WAV, and .SD2 files to be read and written. Conversion to 16-bit is accomplished with high-end dithering which preserves all of the intended richness of your mix. The Multi-band Peak Limiter and Compressor have been enhanced along with a new output stage to allow hard digital clipping, exceptionally smooth tapelike saturation, and everything in between.

While the tests I made with various pieces of music had my head nodding in approval, the real smiles came when I discovered new and highly usable features. One of the best additions is "skins," which provide a chance to customize the virtual rack of equipment to match your mood — everything from "beat up" to "camouflage" and "lizard skin."


T-Racks 24's interface in "lizard skin."

Looking beyond the "visuals," I was pleased to find other useful embellishments. You can check mixes in mono, stereo, and their differences as you would on a high-end console. You can zoom in on the top portion of the level meter to adjust precise output levels enabling very hot signals without clipping. More than 50 factory presets can help you start off in the right direction, and you can adjust and save them at will. You'll also uncover adjustable internal program settings which will allow customization of the program's architecture and sonic characteristics. Users can then share these newly configured settings online through the T-Racks web site.

I've always favored instantaneous feedback and the ability to make unencumbered parameter adjustments. Moving knobs using a mouse is a big pain in the hindquarters. I found myself adjusting too much or too little and playing so much "click — listen, click — listen" that it became distracting. Also, the program initially opens as a half-screen, and after maximizing the window, the rack equipment remains the same size, blackening the unused space around it. The lettering isn't that large to begin with and, unless you like sitting close to your display, there is no chance of making them easier to read. Perhaps IK Multimedia could address this issue in future upgrades.

Overall, T-Racks 24 2.0 is a great product. Many of my tracks have already benefited from its hospitable virtual circuitry, and until they make economical high-end mastering equipment, I'm staying right where I'm at with this agreeable solution. Whether you are creating samples for a Playstation 2 sound bank, composing a momentous CD-ROM game score, or in need of final mastering for a game soundtrack release, this program will work as advertised and at a great price.

✱✱✱✱ | IK Multimedia | T-Racks 24 2.0 | www.t-racks.com

3D GAME ENGINE DESIGN BY DAVID H. EBERLY

reviewed by mark deloura

If the name David Eberly sounds familiar to you, it might be because of his frequent postings to the Usenet newsgroup comp.graphics.algorithms. Or perhaps you've visited his vast repository of source code for graphics and image algorithms at www.magic-software.com. Or it could be that you've examined the NetImmerse game engine, created by Numerical Design Ltd., where Eberly used to be the director of engineering.

Regardless of whether you know of Eberly, his name is certainly one you'll want to remember when you go to your bookshelf to find help on that tricky collision detection problem or inverse kinematics solver. Eberly's recent book, *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics* (Morgan Kaufmann, 2000), is best described as a reference manual for 3D real-time graphics engine programmers. But oh, what a reference manual it is.

Eberly's web site is known by many as a great online resource for code snippets that implement particularly sticky graphics algorithms. But up until this point there haven't been descriptions of the implementations to accompany the code. Now you can find out why each algorithm works, in gory mathematical detail. Eberly has a Ph.D. in mathematics, and it shows — this book is extraordinarily heavy with math. If your linear algebra is rusty, don't start here! Go dust off your old linear algebra texts first. Appendix B does contain some good information on various numerical methods, but you'll need to have a solid math foundation before you delve into it.

Assuming that math doesn't easily scare you, you'll get a lot out of this book. The particularly strong parts include distance and collision detection sections, but there are also good sections on quaternions, scene graphs, curves and surfaces, animation, and terrain rendering. I was especially impressed by a brief dissection of TCB curves (Kochanek-Bartels splines), which I hadn't seen discussed in print before. Algorithm descriptions are accompanied by volumi-

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

nous equations and pseudocode, making this book as much a bathroom reader (bring your pencil) as a quick reference. It would also certainly make a good textbook for courses teaching real-time graphics.

Interspersed every few pages throughout nearly the entire book are references to code files on the CD. The accompanying CD includes reams of code, all of which compiles under both Windows and Linux. It really is an impressive amount of information.

As much as I recommend this book, there are a few caveats. *3D Game Engine Design* frequently gets mired in mathematics. I don't always need to know the complete derivation of the algorithm I'm using, sometimes I'm just looking for a solution, and I need it right now. It can be frustrating digging through all the equations to find this solution. Certainly I expect to find a lot of math when discussing things such as quaternions and parametric curves and surfaces, but frequently I feel like Eberly gives us mathematical proofs as opposed to algorithm descriptions. This needlessly obfuscates the useful techniques described.

3D Game Engine Design doesn't go into detail on practical graphics engine issues such as using hardware transformation and lighting, multi-texture effects, cubic environment maps or shadow-casting. The book is really designed as a high-level game engine theory book for scene-graph-oriented graphics engines, so if what you're looking for are performance optimizations to make your game engine fly, this isn't the right book. But if you're looking to understand how a real-time graphics engine works, or add functionality to your engine, you likely will find what you're looking for.

As with most books, state-of-the-art techniques aren't covered very well. But after you've gone through this book you'll have the necessary expertise to tackle other books, magazines, and the GDC and Siggraph proceedings.

3D Game Engine Design is a very good book that discusses the algorithms behind designing a 3D real-time graphics engine from a mathematically oriented perspective. Check it out.

★★★★½ Morgan Kaufmann | *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics* | www.mkp.com

STARBASE'S CODEWRIGHT 6.5

by scott bilas

Are you an engineer who enjoys working with the least common denominator of editors — the kind that comes with your IDE? Do you accept the defaults given to you by expensive usability labs that cater to rookie Visual Basic programmers? Accept your calling as a true engineer: use a stand-alone editor! I've been using Codewright since version 1 came out for 16-bit Windows years ago. As of this writing, version 6.6 is in the works, and the product has acquired a huge array of features. I doubt I've used even 10 percent of them, so instead I'll focus on the features I rely on heavily.

Codewright comes with everything that you expect from a serious editing tool these days: template expansion, symbol browsing, syntax coloring, version control integration, automatic code formatting, and pointless emulation of VI. It's difficult to break completely free of the IDE, so Codewright provides project and edit window auto-synching. Recently some useful new things have been added, such as an embedded spreadsheet-style XML viewer and partical Unicode support. There are also many tiny features that probably took five minutes to implement but are indispensable. Select a section of code and "filter" it through an external stdin/out-aware command-line app (such as uniq or sort). Use the "edit search path" facility to open files by pattern match from anywhere you please without being forced to browse there through a File>Open dialog or a messy project window. Have all your files automatically saved when you task-switch away. Automatically save a backup every 10 keystrokes or 10 seconds. Check spelling in your comments. Upload via FTP to your web site.

Codewright is all about patterns in text. If you wanted to add support to it for a language it doesn't know about (perhaps one you've written) then it's a matter of configuring the generic lexer to recognize your language's keywords and telling it how to parse comments and such. The symbol browser and many other parts of the editor can be customized using regular expressions. Like other programmer tools, Codewright is also extendable through script. Unlike most programmer tools, you have a choice of three languages: a Visual Basic clone, Perl, and "API Macros" (a C-style scripting language). Or you can do what I do and write the big stuff in an extension DLL, and the little stuff in script. Note that the documentation is not very good, so refer to samples often here. Codewright comes with much of its own source code, which is convenient for über-customizing, hacking, or just figuring out how things work for writing your own macros.

When I'm not editing code, I'm rummaging through it. I probably run a global search every 10 minutes or so when I'm working (I can never remember where anything is). Codewright supports this compulsive habit through named search lists on the multi-file search dialog. You can configure patterns of files using a wildcard with optional path, run searches on them, and then save these lists for later. You can also search and replace from this dialog, even making it modeless if you want. One useful feature is to preserve case on replacement. Probably the single nicest feature of all is that if the file is read-only, Codewright can automatically check the file out from version control, make the changes, and save the file, logging the results to the output window. All of this is optionally prompted, of course.

The main problem I have with Codewright is a problem I have with all editors. There have been no revolutionary advancements in the state of the art for quite some time. Syntax coloring was a leap forward in readability, as it can make a misplaced end quote glaringly obvious. But beyond that decade-old feature, I've found few other new things that make the basic tasks of writing, finding, and modifying code any faster or easier. On the plus side, the makers of Codewright are very responsive and provide excellent technical support.

Codewright is available for Windows 95/98/NT/2000 and is priced at $299.

★★★★½ | Starbase | Codewright 6.5 | www.starbase.com

Return

mark deloura | TALKING TO PEOPLE WHO MAKE A DIFFERENCE

Ed Logg — Super Duper Game Guy

Ed Logg is something of a rarity in our industry; he's an articulate, opinionated developer who has been creating games since the very first arcade hits. Ed has worked on arcade classics, Atari VCS titles, and gorgeous real-time 3D console games. *Game Developer* caught up with him on a cold Friday night to chat about his experiences.

GD. Is that really your title?

EL. I wanted to get some business cards made, and the secretary asked me what I wanted for my title. I said, "I don't care." One of the producers said, "Put 'Super Duper Game Guy.'" The secretary gave me a funny look and asked if I was sure. I said it was fine with me. It makes for a good story.

GD. What are some of the games you've worked on?

EL. My first game at Atari was DIRT BIKE, but SUPER BREAKOUT was doing better in field tests, so we just produced that. Next came VIDEO PINBALL. I did a lot of the vector routines for the vector hardware, which was useful for ASTEROIDS. After ASTEROIDS I did a game called MALIBU, which was a vector-based driving game. That didn't make it into production, either. I did work on CENTIPEDE. After that came MILLIPEDE in 1983. I worked on the original ROAD RUNNER, the video disk version. Then GAUNTLET, and GAUNTLET 2 shortly after that; XYBOTS, SPACE LORDS, STEEL TALONS. I did TETRIS for Tengen on the original NES. Nintendo owes me big time for that one. More recently, GRETZKY HOCKEY, SAN FRANCISCO RUSH, RUSH 2, and RUSH 2049 for Nintendo 64, and RUSH 2049 for Dreamcast.

GD. How is working with Nintendo and Sega in general?

EL. Sega is very good. They have guidelines, and if they're wrong, you can convince them to let you do it your way. Nintendo says, "We have guidelines, and these are the guidelines." You know, these are the rules of God. Thou shalt not change them. That is why we don't have saved ghost races on the N64 version of RUSH 2049.

GD. Where did your get the ideas for ASTEROIDS and CENTIPEDE?

EL. In the case of ASTEROIDS, breaking the rocks apart came from Lyle Rains. He had a game, I think it was called COSMO, and people kept shooting at the rocks in the game, trying to break them up. He decided, why not put in smaller ones so people could shoot and destroy them? I said fine, let's break the big rocks into smaller pieces, to give it some strategy. Then people might not want to just spray bullets around and hit all the big rocks at once. But then we needed something to chase you so that you wouldn't just sit there, so we put in the flying saucer. I also decided to put it all on a vector display, since they used 1024×768 instead of the raster display's 320×240. So out of one meeting with Lyle we came out with vector graphics, breaking rocks into smaller pieces, and a saucer.

CENTIPEDE came from a game called BUG SHOOTER. I don't remember how the centipede character came into it, but I wanted an object that broke into two every time you shot it (like ASTEROIDS). Originally there was a fixed pattern for the mushrooms, and they were nondestructible. Dave Van Elderen, my boss, suggested that you should be able to shoot the mushrooms. I said to myself, if I can shoot the mushrooms I need something that adds mushrooms. So if you shoot the centipede, it leaves a mushroom. Well, now I need something to get rid of mushrooms at the bottom. O.K., the spider will get rid of them, and the spider is the guy that keeps you moving so you don't just sit there. You see the ASTEROIDS model — the spider is my saucer, the centipede segments are the rocks.

GD. How big were your teams on CENTIPEDE and ASTEROIDS?

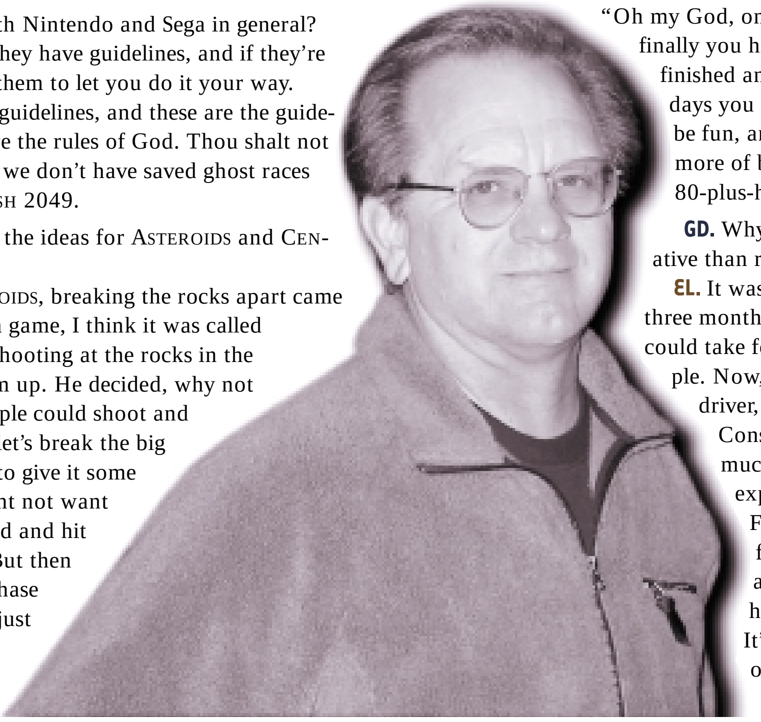
EL. ASTEROIDS was myself, the technician who kept my hardware running, and the engineer that designed the hardware. On CENTIPEDE, Donna Bailey and I did the programming, and there was also a technician and engineer. The graphics I did myself.

GD. Was it more fun to work on games with smaller teams?

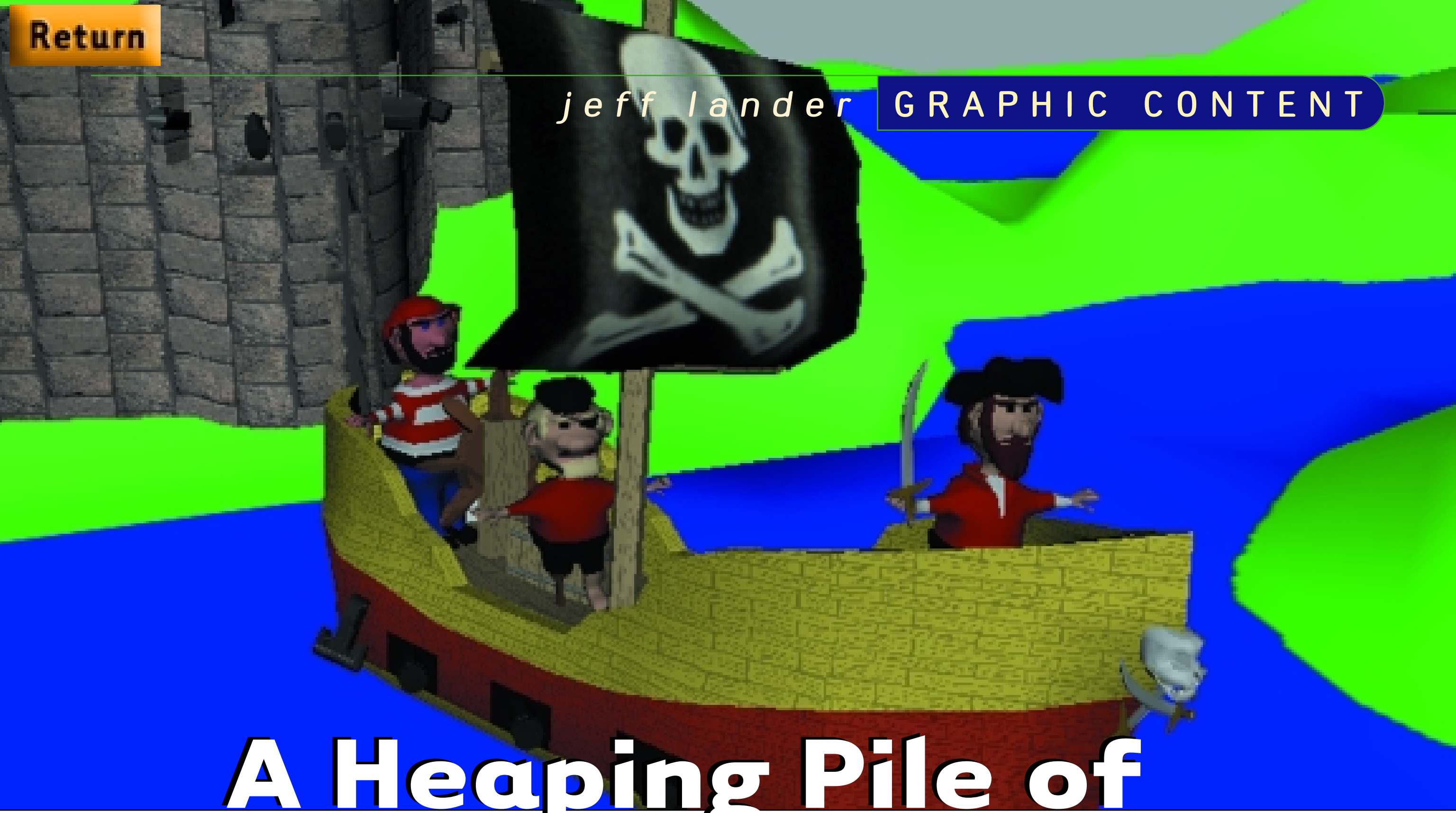
EL. Product development goes through several phases. You have maybe three months of euphoria until the next stage, which I call the grind-it-out stage. The third stage is the panic stage, "Oh my God, only a couple months left." And finally you hit the worry stage when you're finished and you hope it sells well. Nowadays you get going, it looks like it's going to be fun, and then you have 12 months or more of boredom followed by months of 80-plus-hour weeks.

GD. Why do older games seem more creative than recent titles?

EL. It was a lot easier when a project took three months, and if it died, so what? You could take four shots a year, with fewer people. Now, in coin-op, if it isn't a shooter, driver, fighter, or sports game, forget it. Consumer titles these days cost so much to develop, and everyone expects movies, everyone expects FMV. Now there are links with films, and more licensing of characters. It's try anything to get some hook into the consumer market. It's no wonder more time is spent on things other than gameplay.


ED LOGG. Game industry pioneer.

Return



A Heaping Pile of Pirate Booty

Game programmers like to think of themselves as a rebellious lot. I like to think that we put a little of our own alter egos into the programs we write (assisted of course by talented artists who make the characters come to life). Although we programmers are not plundering and pillaging, the characters in our games probably are.

In last month's column ("Graphics Programming and the Tower of Babel," March 2001), I dug into the topic of vertex shaders. These small but powerful programs give a programmer direct control over the transformation pipeline of a 3D vertex. Along with that power comes a bit of confusion and uncertainty. What level of support for these features will individual graphics cards actually have? Will software fallbacks exist for situations that the hardware is not capable of handling? How does art production deal with these variables? As I am writing this, I do not even have a graphics card capable of executing these vertex shaders in the hardware. Fortunately, the software implementation of DirectX 8 allows developers to experiment with the concepts and try various techniques even before the hardware is available. That way, when the hardware arrives and management wants to see the game with the new graphics features, the programming staff won't be forced to walk the plank.

Yo Ho Ho and a Bunch of Bones

When I left off last time, I had implemented the basic cartoon-cel-style painting with a vertex shader using DirectX 8. This simple program took the vertex position and transformed it into the position to be rendered. The vertex shader also transformed the vertex normal and calculated the dot product of the normal and the light vector to determine the shade to select from my 1D shade texture. You can refer to the final vertex shader in last month's column.

This shader was sufficient to render static objects with a flat-color shading style. Since there are many times that this may be exactly what I need, I will certainly keep this shader around. That's one of the good things about using programmable vertex shaders. Developers can create a variety of useful effects and keep them in a library for use when needed. Of course, the downside to this is the need to create a bunch of shaders for the various effects needed in a typical game production. Hopefully, graphics card manufacturers, Microsoft, and individual developers will help with this problem by continuing to share code and ideas.

The goal when I began this task of playing with DirectX 8 was to get my character animation system up and running with the new hardware features. The simple vertex shader from last month's column is not up to the task of working with animated characters. It is much too simple for anything that complicated. To get any serious animation done, the vertex shader is going to need to get beefed up a bit.

DirectX 8 has quite a few options when it comes to character animation. Along with support for morphing between multiple vertex streams, the API specifies several methods for matrix deformation of vertices. This is quite a change from the past, when everything had to be done in the application program. However, it means I

JEFF LANDER | *When not hoisting the mainsail and embarking on exciting adventures around the world, Jeff can be found manning the bilge pump at Darwin 3D. Send mail to the scurvy wretch at jeffl@darwin3d.com.*

need to make some choices.

I am going to have to assume at this point that you are at least somewhat familiar with animation by matrix deformation — I know I've talked about it a few times in this column. The technique, often called skinning or bone-based animation, uses a series of transformation matrices, each possibly contributing to the final position of a vertex. The matrices are often used for character animation by arranging them in a skeleton hierarchy with each matrix representing a bone in the character's body. If this technique is not familiar to you, I suggest you take a look back at my article "Over My Dead, Polygonal Body!" (Graphic Content, October 1999) for a quick overview of the mathematics of the technique.

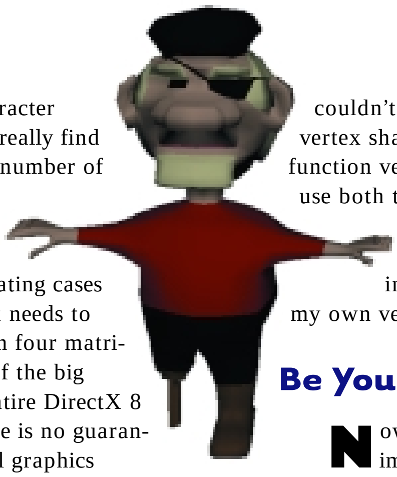
DirectX 8 specifies several fixed-function pathways for matrix deformation of vertices; by this I mean methods that do not require the use of programmable vertex shaders. One method is to allow several matrices to be specified as influencing the vertices of a triangle. For each vertex, a weight value is submitted for each matrix that influences it. These weights determine the amount of influence the corresponding matrix has on the final transformed position of the vertex. The number of matrices that can be specified for each vertex is limited by the hardware implementation and can be determined by querying the hardware. Since a small number of matrices are specified per vertex, it is most likely necessary to divide up the model into groups of triangles that share matrix sets. In a model with a large number of matrices in the skeleton, this could mean breaking up the model into a bunch of pieces.

A second method introduced with DirectX 8 is the use of indexed matrix deformation, called "indexed vertex blending" in Direct3D-speak. In this method, a larger number of matrices are specified in a table. Each vertex is still influenced by a small number of matrices, with the maximum number specified by the hardware. However, in this case the matrices used are selected from the larger table or palette of matrices. This is confusing, so let me use an example. The DirectX 8 software implementation of the fixed-function pipeline allows for up to 256 matrices to be specified for use in deformation. Each vertex can be influenced by up to four of these matrices as specified by an index value in the vertex structure. To make this work, each vertex has a structure similar to:

```
struct sample_vertex
{
   // Position XYZW
   float   pos[4];
   // Amount of influence each matrix has
   //      w4 = 1 - (w1+w2+w3)
   float   weight[3];
   // Index to matrices [0-255] influencing
   //      this vertex
   byte    index[4];
   float   normal[3];
};
```

This is a pretty flexible system. The palette of 256 matrices is probably enough for most character applications. I don't really find that the limit on the number of matrices that can influence a vertex is very restrictive. I have a hard time creating cases where a single vertex needs to blend with more than four matrices. Of course, one of the big problems with the entire DirectX 8 approach is that there is no guarantee that an individual graphics card will support a 256-matrix palette in hardware. To determine the true number of matrices supported, the program needs to query the hardware and check the MaxVertexBlendMatrix setting. If the system does support all 256 matrices, this is a pretty good setup for animation.

However, when you choose to use the fixed-function pipeline for transformation of vertices, you lose all the benefits of using vertex shaders. For example, I couldn't do my fancy cartoon-ink vertex shader along with the fixed-function vertex pipeline. In order to use both techniques together, I will need to duplicate the functionality of the indexed vertex blending in my own vertex shader.


Low-polygon pirate.

Be Your Own Boss

Now that I have decided to implement the indexed vertex blending system with a programmable vertex shader, the fun can really start. I first need to determine the capabilities of the hardware I am running on. Determining the amount of space for matrix information is the first crucial step. This will determine how many matrices I can have in my blend table. There are two ways to go about getting data to a vertex shader. Data can be passed in through the vertex structure or put into the constant registers. It does not make any sense to me to pass the matrix data in the vertex structure, since the same matrix may be used for multiple vertices. Therefore, the matrix data must be put in the constant register space. The other thing I will need in order to handle indexed vertex blending is support for indirect accessing of this constant register space. DirectX 8 provides for this capability in the form of the address register, `A`. In version 1 of the vertex shader spec, there was no support for this address register. In version 1.1, there is one address register, labeled `A0.x`. Since I need this support, I will have to check the VertexShaderVersion and make sure it is at least 1.1. If the hardware doesn't have this support, I can use the software vertex pipeline.

As I said last month, DirectX 8 hardware that supports vertex shaders must have at least 96 constant registers. It can have more than 96, but that is the minimum support. The number of constant registers supported is determined by checking the MaxVertexShaderConst capability setting.

Let me work with the base case of 96

registers and a single address register. This setup should be acceptable for my character animation system. The way I transformed a vertex and normal using the vertex shader last month was by placing the four rows of the transformation matrix into four constant registers. I used four more constant registers to store the four rows of the inverse transpose of this matrix for use in transforming the vertex normal. That is a total of eight registers per matrix, giving me a theorerical maximum of 12 matrices in the matrix blend table (12 × 8 = 96). That doesn't include the register I will need for the lighting vector or anything else, so it really doesn't give me much power.

I could dump support for vertex lighting, since that would double the available space, but that really isn't a good option. When transforming the normal, I really only used the first three rows, so I can save one register per matrix that way. But in order to really free up space, I need to do something drastic.

Trash the Transpose

I never really have explained why I need the inverse-transpose matrix for lighting. The projection-transformation matrix that transforms the vertices into projection-space coordinates ready to be clipped and sent to the screen is probably not orthogonal. That means it usually consists of rotations, translations, and a perspective scaling function. To rotate a normal into the correct space for lighting, the normal needs to be transformed by the inverse transpose of the projection-transformation matrix.

For my character animations, I animate most of the bones by simply rotating the bones. Since I am not planning on having bones dynamically scale, the bone matrices contain only rotations and translations. For a simple rotation matrix, the inverse of the matrix is equal to the transpose. That is:

$$M^{-1} = M^T$$

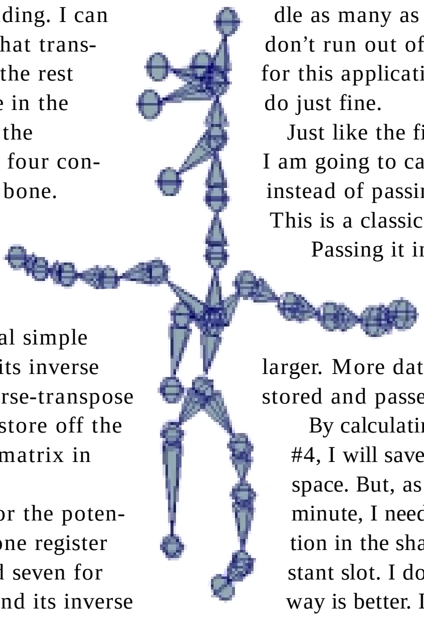
therefore

$$(M^T)^{-1} = M$$

I can use this fact to reduce drastically the number of constant registers needed for indexed vertex blending. I can calculate the matrices that transform the vertices from the rest position to world space in the deformed pose. This is the matrix that is stored in four constant registers for each bone.

To take this deformed world-space pose and project it into projection space, I will need a final simple projection matrix and its inverse transpose. For the inverse-transpose matrix, I only need to store off the first three rows of the matrix in three constants.

What does that do for the potential capability? I have one register for the light vector, and seven for the projection matrix and its inverse transpose. That leaves 88 registers for the bones. At three registers per bone matrix, I can deform from a palette of 29 bones with one register left over. Not too bad.


Skeleton for the pirate with 29 bones.

I can now set up the vertex structure for this system. I need the vertex position, normal, UV coordinates (used for the shading as well as texturing), vertex weights, and matrix indices. This is pretty much the same as the fixed-function pipeline I described above.

```
struct sample_vertex
{
  // V0.xyzw       = Position
  float pos[4];
  // V1.xyz = Amount of influence each
  //       matrix has
  //       w4 = 1 - (w1+w2+w3)
  float   weight[3];
  // V2.xyzw = Index to 4 registers
  //       where each matrix is stored
  byte    index[4];
  // V3.xyz
  float   normal[3];
  // V4.x   = Shade texture coordinate
  float   u0;
  // V5.xy  = Mapping coordinates
  float   u1, v1;
};
```

Since I am rolling my own deformation system, I am not really limited to blending only four matrices per vertex. I could handle as many as I want as long as I don't run out of instruction space. But for this application, four weights will do just fine.

Just like the fixed-function pipeline, I am going to calculate weight #4 instead of passing it into the shader. This is a classic developer trade-off. Passing it in the vertex structure will mean the vertex structure size is one float larger. More data will need to be stored and passed over the bus.

By calculating the value for weight #4, I will save a bunch of vertex data space. But, as I will describe in a minute, I need to sacrifice an instruction in the shader as well as one constant slot. I don't really know which way is better. It probably will depend on the application, so you should try doing both and see how it works out.

The bone deformation matrices need to be laid out such that I can access them easily in the vertex shader. Each matrix takes three slots, so the logical layout is:

Constant n	= Matrix1 Row1
Constant $n + 1$	= Matrix1 Row2
Constant $n + 2$	= Matrix1 Row3
Constant $n + 3$	= Matrix2 Row1
Constant $n + 4$	= Matrix2 Row2
Constant $n + 5$	= Matrix2 Row3
Constant $n + 6$	= Matrix3 Row1

This is done for all 29 matrices. You will notice that since each matrix occurs every three registers, I will need to set the matrix indices in the vertex structure accordingly. For example, if a vertex is influenced by matrix #20, I will store the value 20 × 3, or 60, as the index for that matrix. I now have all my data laid out and I am ready to write the vertex shader.

Cast off the Mooring Lines and Set Sail

First I need to declare the shader version number:

```
; Declare the shader version number
vs.1.1
```

I want to deal with the first bone matrix immediately. I will start by getting the index offset and putting it in the address register.

```
; get the first matrix index
mov a0.x, v2.x
```

I can then transform the vertex position by the first bone matrix using the four-component dot-product operator. This transformed position is multiplied by the weight and stored.

```
; multiply by matrix 1
dp4 r0.x, v0, c[a0.x + 0]
dp4 r0.y, v0, c[a0.x + 1]
dp4 r0.z, v0, c[a0.x + 2]
; factor weight and store in reg 0
mul r0.xyz, v1.x, r0
```

I now need to do the same thing for the vertex normal. Notice how the normal only uses the first three columns of the matrix and thus the `dp3` instruction.

```
; multiply normal by matrix 1
dp3 r2.x, v3, c[a0.x + 0]
dp3 r2.y, v3, c[a0.x + 1]
dp3 r2.z, v3, c[a0.x + 2]
; factor weight and store
mul r2.xyz, v1.x, r2
```

For each subsequent bone, the result of the transformation operation is multiplied by the weight and added to the total result using a single `mad` (multiply and add) instruction. For example, to handle the position with the second bone matrix:

```
; get the second matrix index
mov a0.x, v2.y
; multiply by matrix 2
dp4 r1.x, v0, c[a0.x + 0]
dp4 r1.y, v0, c[a0.x + 1]
dp4 r1.z, v0, c[a0.x + 2]
; multiply weight and accumulate result
mad r0.xyz, v1.y, r1, r0
```

This process continues for the rest of the second and third bone matrices. For the fourth matrix, the vertex weight needs to be calculated. To do this, I will utilize a method that is used in several of the Microsoft and Nvidia Direct3D sample applications. I can store the values (`x = 1, y = -1`) in my remaining empty constant register. A constant register can be fully swizzled, that means the contents of any component of the register can be swapped to the other. Using this fact, I can calculate the weight formula with two shader instructions.

```
; store off the weights temporarily in r4
mov r4, v1
; calculate w4 = (1 - (w1 + w2 + w3))
dp4 r4.w, r4, c[8].yyyx
```

See how that works? Matrix #4 is then blended in the same way as the others.

The last thing I need to do is transform the final position by the projection matrix and the normal by the inverse-transpose projection. The vertex is then shaded just like last month, and I am done. I now have a shaded vertex that has been transformed by a weighted blend of four matrices from a palette of 29. The grand total of instructions for the shader is 49, out of a possible total of 128 instructions.

Potential Leaks in the Boat

The system works very well. But there could still be a few problems. On the basic DirectX 8 hardware with 96 constant registers, I can only have 29 bones in the palette. This is far fewer than the 256 matrices allowed in the fixed-function vertex pathway. What is the reasoning for this? It sure seems like the number 96 came out of nowhere. It probably is the number of registers that the first hardware designed for the shader system had. This same number appears in the recently released OpenGL extension NV_vertex_program by Nvidia. Some hardware may support more registers. If so, the system I describe will still work pretty well, but you may have problems storing the index values in a byte if you support more than 84 or so bones.

When there is not support for the number of matrices needed for a model, the easiest option is to break the object up into subobjects where each subobject needs fewer matrices. A good solution for characters may be to have 29 matrices for the body and 29 more for the head and face. Then you only need to have two draw calls per character. As an aid to development, Matrox has developed a tool that allows artists to break up an object into subobjects automatically based on bone and weight settings.

Another problem occurs when you start thinking about using the subdivision methods in DirectX 8. There is support for automatic subdivision of polygons to allow for smoother curved surfaces and such. While this works well for general polygonal objects, it is not clear how this can work for vertices that are deformed by matrix blending. How do you interpolate a series of matrix weights or matrix indices across the surface of a polygon? For now, I think you will need to forget automatic subdivision if you want to use indexed vertex blending. You can still use software methods to tessellate the object before submitting it. But then you lose the benefit of passing small geometry over the computer bus.

A further potential problem is that the next thing in graphics may be the use of displacement maps. How will these maps integrate with the matrix blending techniques? All of this is a little unclear right now. However, with that all said, there clearly will be strong support from hardware makers for DirectX 8–capable hardware. These programmable vertex shaders will most likely also be useful for the Xbox console system. Since the technique I described in this column will clearly allow detailed characters to be animated directly using the transformation and lighting hardware on the card, I think it is well worth it to add support for shaders like this to your next game project.

You can get the final vertex shader program as well as a sample application off *Game Developer*'s web site at www.gdmag.com. I would like to thank Juan Guardado of Matrox and Cass Everitt of Nvidia for discussing the issues with me.

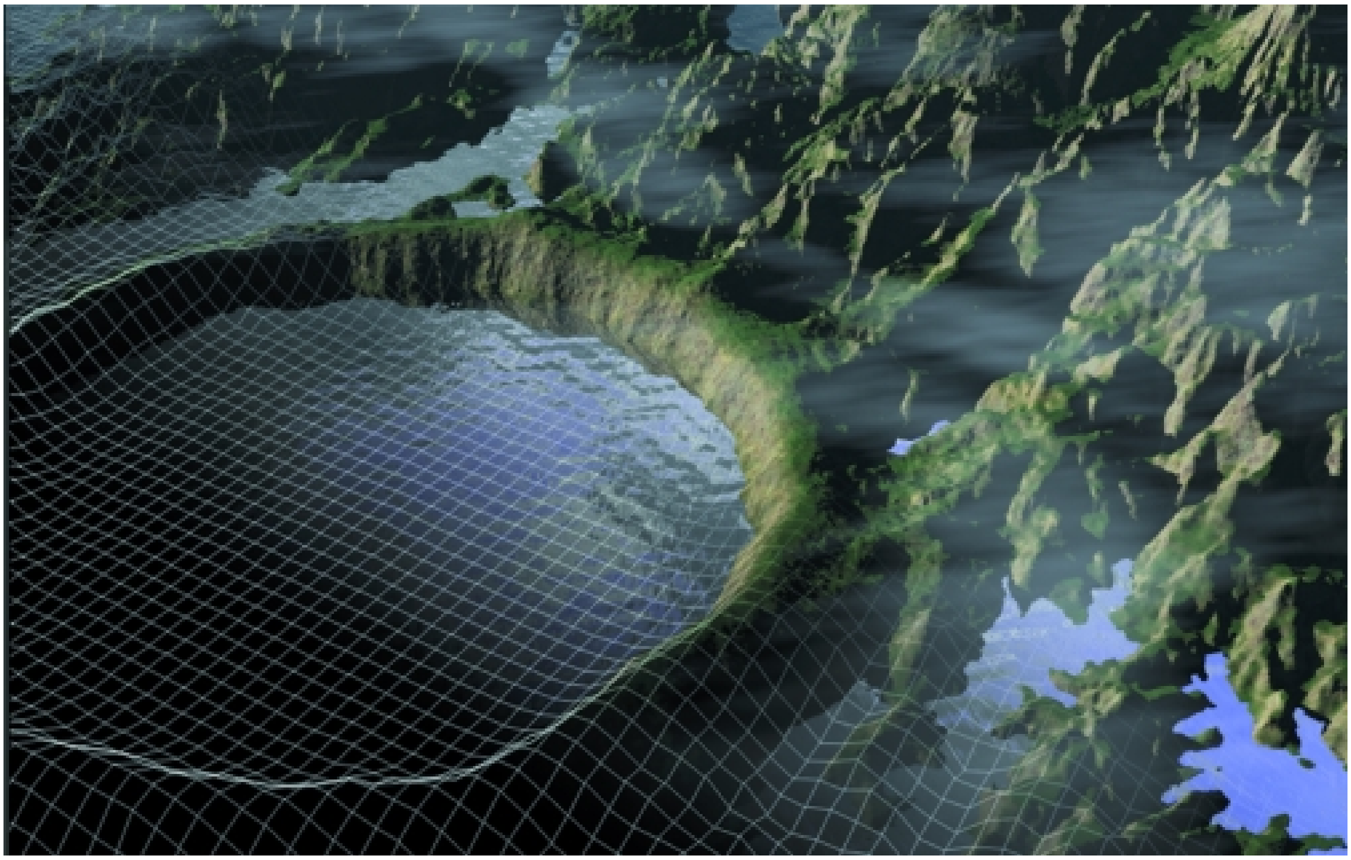
FOR MORE INFORMATION

DirectX 8.0 Development Kit, Documentation, and Sample Code
www.microsoft.com/directx

Nvidia DirectX 8.0 Sample Code and Developer Information
www.nvidia.com

Matrox Developer Relations
www.matrox.com

Terraforming, Part 2



This month's column will be a continuation of the tutorial on terraforming we started in last month's column ("Terraforming, Part 1," March 2001). We are using both Bryce and Photoshop to create realistic terrain elements. Based upon the desired output, the mesh can be modified at export for optimal usage. By utilizing the grayscale terrain editor in Bryce and Photoshop, we can have a high degree of control in creating our terrain.

Creating a Tiled Sequence

A number of the more recent real-time 3D games are using actual mesh geometry as tile pieces. In previous years, these were simply rendered bitmaps that ended up being tiled in order to create the final game terrain. More recent games are starting to push toward the use of 3D modeled terrain elements in real-time 3D engines. This shift toward a 3D environment keeps the terrain artist on the cutting edge of the technological knife. With the power that is available on systems now, it becomes more realistic to consider fully modeled terrains. Utilizing Bryce 4 as the basic terrain maker, we will explore how to extract these elements in a usable way that might be the first step in creating some terrain nodes.

When you first fire up the terrain editor in Bryce and hit the fractal button, you'll see the standard mountain displacement map replaced with a flat terrain piece. While this looks nice, you may find the need to have greater resolution in the image or a different scale. Changing the Grid (located in the middle of the interface) to a higher terrain resolution doesn't have an effect on the scale of the displacement map. What you need is the ability to take a given terrain piece and create a number of matched sets that fit together seamlessly with one another. While this seems to be a tall order, there are some nice features in Bryce that are designed with just this in mind.

The first step is to determine how many mesh nodes you want to create. For the sake of argument, let's say you decided upon a 3×3 grid (nine individual pieces) that will make up your base ground terrain. With that in mind, it's easiest to set up the first grid piece and then repeat it, changing the fractal pattern as you

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

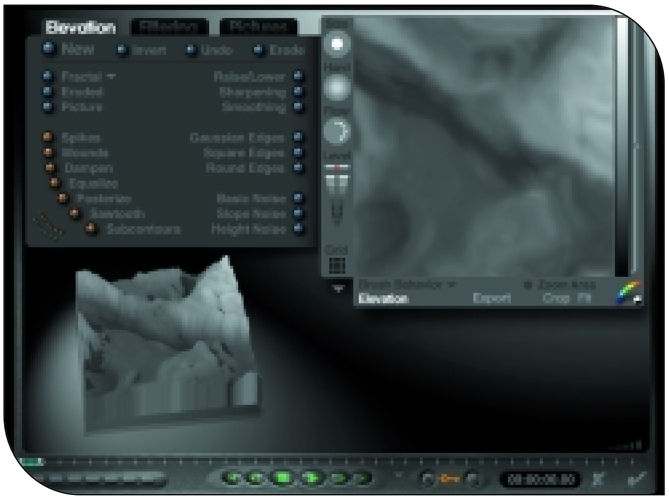


FIGURE 1. An example of a lava fractal terrain pattern.

do so. This makes it easier to see errors, and to see the progress as you go.

A seamless texture in Bryce is exactly the same as a tiled 2D image with the addition of height information. The first step is to create a standard mesh in Bryce by clicking on the small mountain icon in the Create menu (Create>Terrain). This will result in a mesh being added to your scene with the default 128×128 grayscale displacement map assigned to it. The terrain will automatically be selected when it enters, and will have a set of small icons floating to the right of its bounding box. Select the "E" icon (Edit Object) and you will find yourself in the grayscale Terrain Editor.

Now it's time to do some experimenting with the fractal types. Go to the Fractal pull-down and select a fractal pattern. After selecting one, you won't see any change until you click on the Fractal button. Make sure you click on it several times, as the settings are randomized with each click, which can result in fairly different looks. The trick here is that it's a one-shot deal. There's no undo function to go back to the texture before your current one. Even though Control-Z will undo the grayscale image you see, it doesn't undo the fractal pattern generated in memory by Bryce, so you won't be able to duplicate it. Don't worry too much about this, since once you settle on a fractal type, it's fairly easy to find an acceptable-looking one with a few more clicks. Spend some time testing out the variations, as they are quite striking.

For a unique-looking world, I decided to go with the Lava Fractal, and eventually found a pattern I liked (see Figure 1). Once you've decided on an image, go to the Fractal pull-down menu again (you'll do this three times) and remove the check marks for Random Extent, Random Position, and Random Character. These lock the pattern down so that when you click on the button again, you'll get the exact same result time and again.

What are the functions of these selections you just turned off? Well, Random Extent is essentially the scale or zoom factor you are seeing for the terrain piece. Random Position is just that, the position of the fractal pattern. And Random Character deals more with the granularity or effects of the pattern, among other things.

Now that all three are off, go to the Grid button and select a smaller resolution size from the selections. For this tutorial, I chose 64×64. Click on the Fractal button again, and it will rerender the same pattern but at a lower resolution. We could have just as easily gone up in resolution, which might be desired if you are targeting output that won't be put into a real-time environment. Since our target output will be grid pieces that will need to be optimized anyhow, it's overkill to crank up the resolution. A 64×64 grid terrain piece is still 1,152 polygons in size, so going to a higher displacement map won't accomplish much besides slowing your process down.

Now that the single grid is set, it's time to start the duplication process. What we will be doing is duplicating the mesh pieces, shifting them into place, and reapplying the fractal effect in a tiled method to create a seamless match.

The first step is to go into the main Bryce application and set the viewport to top-down by sliding the Control>View icon to the top-down image. Once there, you will see your terrain piece, the camera, and an additional ground plane that defaults into the scene. Select the ground plane and delete it. It will be the larger of the two grids in the scene.

The next step is to set the material on your terrain piece to a flat gray instead of the texture Bryce uses automatically. This will dramatically speed up the renders by simplifying the texture Bryce is using. Click on your terrain mesh and select the small "M" icon (Materials Lab) which will put you in the Texture Editor. Now, select the small arrow to the right of the preview window by clicking on it once. This will bring up the Materials presets window. From here, select Simple & Fast, then choose a color. Flat gray works well for this tutorial. Click on the O.K. button and you will end up back at the main Bryce interface.

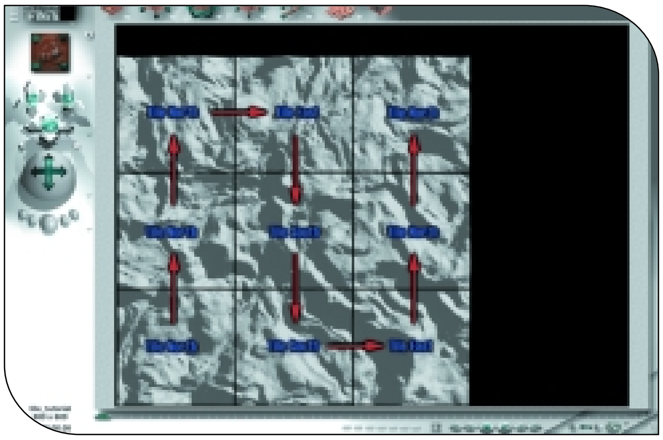
The last default we want to set is the sky color. Since it will try to render a default cloud set with haze and a bunch of other things that we don't need for our tutorial, we need to turn the sky off. To do this, go to the far left icon of the sky settings. It is located directly below the word Create. Click on the down arrow, and select Atmosphere Off.

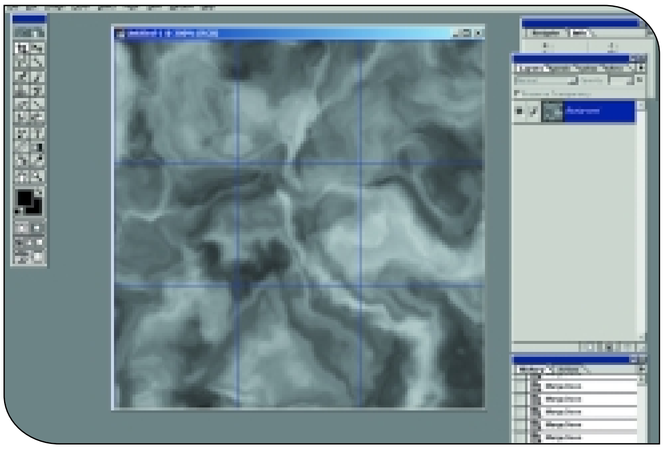
Now we need to get our terrain in the right location for tiling. Press the spacebar on your keyboard and hold it down. This will turn the cursor into a hand, which will let you place the grid in the bottom-left part of the view. Alternatively, you can go into the document setup area and define a larger resolution for the screen.

Since our target output will be reduced in resolution quite a bit, it's a good idea to increase the height for the grid so that our mesh is more distinct. This is sort of a relative thing, open to artist choice. Go to the mesh and click on the "A" icon (Attributes). In the bottom of the window, select the Size Y box, and double the value there.

We will need to duplicate the terrain grid, starting with the bottom left of the screen. Do this by selecting the shortcut Control-D. Bryce will duplicate the terrain and place it in exactly the same location as the original. Without clicking anywhere else, press the Shift key and the up arrow eight times. This will put it close to the next grid position. A quick render here will show two identical grids.

The next step is to tile the displacement map being applied to the second mesh terrain piece. So select the newly created mesh,





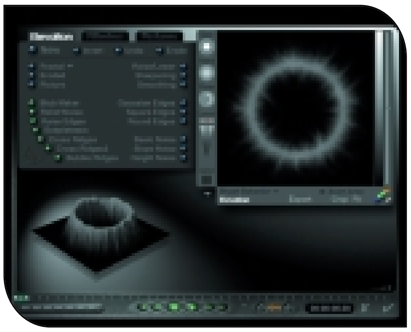


FIGURE 2 (top). Following a zigzag pattern while tiling keeps the terrain seamless. FIGURE 3 (middle). The 3×3 grid of displacement maps transferred over to Photoshop. FIGURE 4 (bottom). The hole applied to the terrain mesh.

and once again, click on the Edit button to go into the grayscale editor. Go to the Fractal pull-down menu and go down to Tile North and select it. This will enable it with a check mark. Now, click once on the Fractal button. The pattern will change and be applied to the second mesh. Click on the O.K. button and verify your new tiled piece by doing a render. You will see that the pattern is perfectly matched with the next piece.

Repeat this process again by selecting the newly altered terrain mesh, duplicating it, and shifting it up eight spaces. Tile the fractal pattern again by repeating the process of going into the editor and clicking on the Fractal button. Your previous setting of Tile North will be applied to this new piece, and again, you will have a perfectly matched piece. Render once more to verify that everything is progressing as expected.

Next, duplicate the mesh, and holding down the Shift key, press the right arrow eight times to move the new piece to the right. In the grayscale editor, go to the Fractal pull-down menu, and select Tile East, which will change the direction you are tiling. Now click the Fractal button, and you will get a perfect match to the right. (If you get out of sequence by tiling the wrong direction, you will need to start over. The process is sort of a one-way street.) Repeat these steps going down the column, but be sure to select Tile South. Continue this process until your grid is complete (see Figure 2).

Altering the Displacement Maps with Photoshop

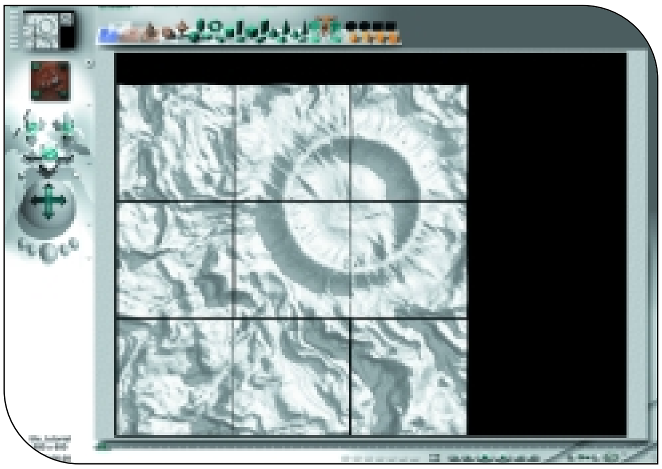
The next thing we want to do is give a few of these terrain grids some customization using Photoshop. As long as you keep everything on a grid, and are careful, you can alter the images without too much concern.

The first step is to set up a Photoshop file to the correct size. Since we have a 3×3 grid, each being 64 pixels across, we need to create a file that is 192×192 in Photoshop. Once this is done, set your grid to correspond with the mesh sizes. Go to Preferences> Guides & Grid. Set the grid line to 64 pixels with the subdivision set to 1. Now, when we copy and paste the files into Photoshop, they'll snap to the grid you've established.

Using the method described in last month's column, open up both applications and use the cut-and-paste method to transfer each displacement map over to the Photoshop file in its correct place (see Figure 3). Once all nine pieces have been transferred over, collapse the entire stack, and you will end up with the displacement maps that were used to create each Bryce terrain mesh.

The next step is to create a unique terrain feature. In this case, I decided on an impact crater. The easiest way to do this is to start with a new mountain mesh. Next, go into the grayscale Terrain Editor and click on the small set of dots in the bottom-left corner of the Elevation menu. This toggles some of the Elevation effects on and off. Do this until you see the Blob Maker effect, and click once on it. This will create a large, evenly shaped mound. Now go to the far right of the editor. There you will find a gradient bar beside the terrain image with a bracket by it. The bracket is movable and allows you to set the clipping height of the mesh. Select the bottom part of the bracket and slide it up until you have clipped the lower portion of the mountain off. It will display as a red area. Now, click and hold the Raise/Lower button, and then slide it to the right. This should turn the active area on the grayscale image black. Go back to the gradient bar and slide it back down. You should now have a hole cut into your terrain mesh. Apply some erosion to the image, and you end up with a fairly believable crater that didn't take too long to make (see Figure 4).

Once you have your impact crater complete, copy the image into the clipboard using the Control-C shortcut. Switch to Photoshop and paste the grayscale image into your 3×3 grid. With a little layer work, the crater can be added to the existing texture map easily. The exact placement is up to you. Since the export will take the new geometry into account, you need not concern yourself



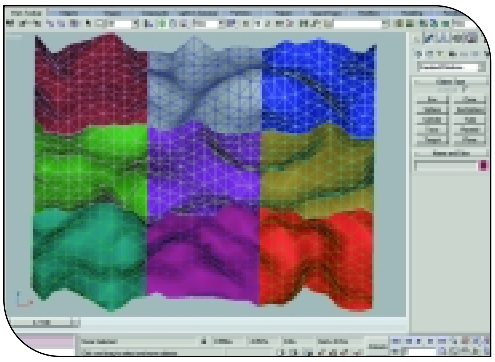


FIGURE 6 (above). A crater merged into displacement maps and rendered in Bryce. FIGURE 7 (left). Grid pieces reduced in polygon count and imported into 3DS Max.

with the image crossing over a seam. Once you are done placing the crater, collapse all the layers down so that you are dealing with only one grayscale image.

Set the marquee selector in Photoshop to rectangle. Go into the Options menu and select Fixed Size from the Style pull-down and enter 64 for both the width and the height. The last step is to make sure the mode for the Photoshop file is set to RGB mode, not grayscale. For some reason, Bryce doesn't like grayscale images in the clipboard and will react with an "invalid clipboard format" error message.

Now, reverse the process you went through at the beginning of this section to copy the Photoshop files back into the grayscale displacements for each mesh in Bryce. Once you are done, render the file to ensure that it all went according to plan (see Figure 6).

Exporting Your Meshes

Now that you have your nine meshes ready for export, let's take a look at the export utility in Bryce 4. There are two ways to access the exporter. First, after selecting the first mesh you created, simply use the top tool pull-down interface and select File>Export Object. The second way is to use the exporter from within the grayscale editor. Once you are in the editor, you can select the word Export located directly below the grayscale image. Either way, you end up with a menu that lets you choose the file format you need. In this case, I'll export in .3DS format for use in 3DS Max. When you initially created the meshes, Bryce named each new terrain piece in sequence, so you shouldn't need to rename the output files.

Once you select the file format, you'll end up in the Terrain Export Lab. Within this menu, you have the capability to change the mesh resolution before you export. In addition, you can select the method that Bryce uses to alter the level of detail when reducing the polygon count. The two types are Grid Triangulation and Adaptive Triangulation. The Grid method keeps your polygons on a consistent grid, which is what we will want for our output. Adaptive Triangulation optimizes based upon polygon angle and detail. Larger areas that are uniform and low detail (a flat plain, for example) will reduce substantially, while rugged areas with a lot of height variation will have more detail. Though Adaptive is more efficient, it doesn't maintain the integrity of the grid edge from texture piece to texture piece. Since we are interested in edges that are matched to one another, we need to choose the method that doesn't eliminate what could end up being critical vertex points. The exporter automatically defaults to Grid Triangulation, which is what we need.

The mesh originally created had a grid resolution of 64×64. In Bryce, this equates to a terrain with a resolution of 1,152 polygons. This is probably overkill for most real-time game applications, so we will want to reduce it down to a reasonable number. The slider at the bottom of the Terrain Export Lab allows you to increase or decrease the grid resolution interactively while watching the results in real time. To the right of the slider is a numerical output of the polygon count. For this tutorial, I selected an output of 200 polygons per grid. Select and export each mesh grid you created with 200 polygons as your target size.

Importing the Mesh Pieces into 3DS Max

Now that all of the meshes are exported, open up your 3D program of choice. In this case, I've used 3DS Max. To import .3DS format files, you need to use the File>Import command and select the first terrain. The importer will ask if you want to merge the new item into the current scene or create a new one. Choose Merge and also choose Convert Units so that the mesh comes in at a reasonable size rather than tiny. You can easily scale the resultant imports, but I find it easier to convert the units. One of the nice things about this process is that the export from Bryce remembers its offset so each imported file loads in at the correct spot in relation to the other grids. Continue to load each grid into Max until all nine are in place.

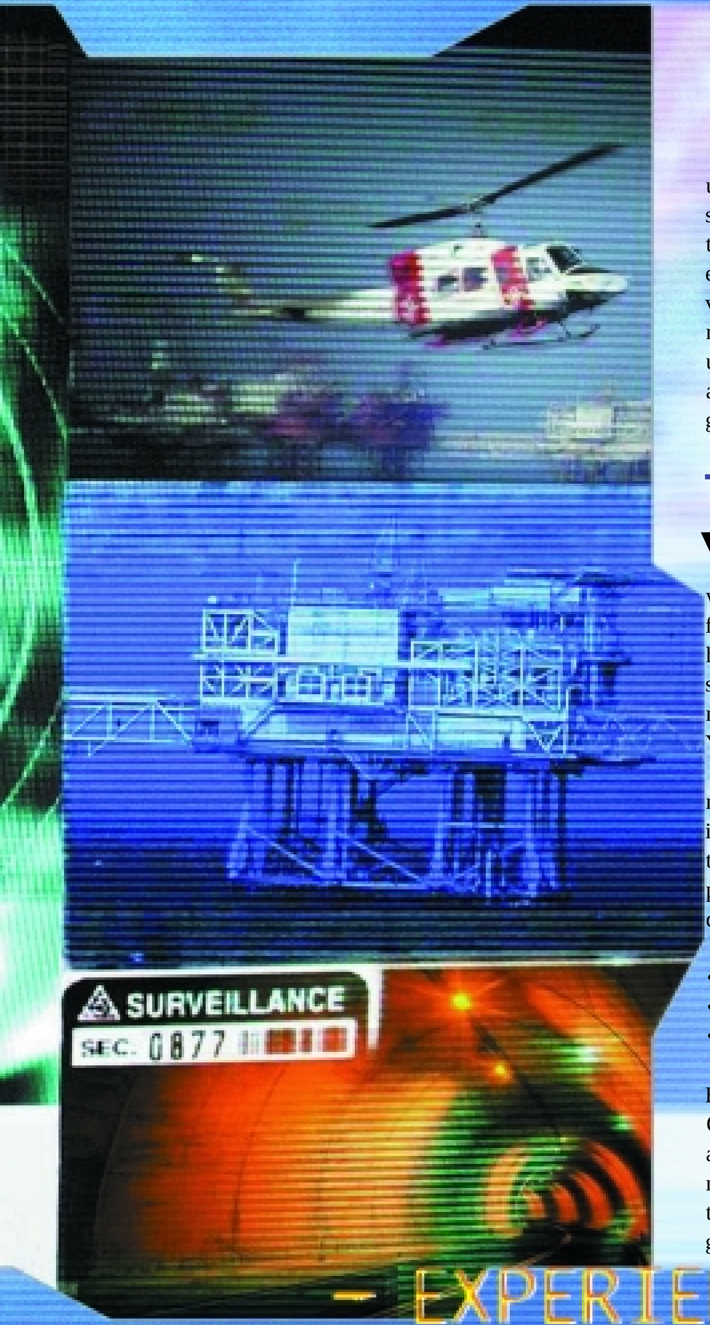
With all nine grids in place, you can see the results of your work (see Figure 7). While converting them down to 200 polygons per grid means that you lost a lot of detail, you will find that you can make up for it with smart texture mapping. Speaking of texture mapping, you may have noticed all those settings in the exporter that related to texture maps. While Bryce has the capability of exporting the procedurally generated texture, it can be challenging to re-create. There are better ways to get the texture out of Bryce for use in other applications, but that's for another column down the road. Hopefully, this tutorial has given you some ways to add Bryce to your war chest of software packages.

Return

THE MAKING OF MAJESTIC

NEIL YOUNG, RALPH GUGGENHEIM, AND RICH MOORE

NEIL YOUNG | *Neil is the creator of MAJESTIC, which started officially in May 1999, when he relocated to the Bay Area from Austin, Tex. Before joining EA, Neil had been the general manager of Origin Systems.* **RALPH GUGGENHEIM** | *Ralph is one of the founding members of Pixar and the producer of the movie* Toy Story. *Ralph is the executive producer for MAJESTIC.* **RICH MOORE** | *Rich has a deep understanding of games from his many years at Atari Coin-Op and Capcom and also of the Internet from his time as a technical leader at MediaShower. Rich is the director of technology for MAJESTIC.*



What if you could be at the very center of your own suspense thriller? What if the characters called you on the phone, left clues on your fax machine, and infiltrated your life? What if you couldn't discern between fact and fiction?

This is MAJESTIC, an episodic online entertainment experience. It's an unfolding mystery adventure that uses the Internet as the canvas for its story, weaving the player through both real and fictional experiences in real time. Highly personalized and naturally paced, MAJESTIC tailors the experience specifically for each participant in order to immerse the player at the very heart of a developing conspiracy. MAJESTIC players assume the leading role at the center of their own thriller, interacting with other characters, uncovering clues, searching for answers, collecting and using digital objects, and resolving challenges. In many ways MAJESTIC is a classic adventure game, albeit with a dramatically different set of locks and keys.

The Big Idea

When we founded Synthetic (the Electronic Arts studio that is developing MAJESTIC), we set out to build much more than MAJESTIC — we wanted to create a new type of entertainment, something that was built for the generation that had grown up with interactivity, but that had perhaps grown bored with just developing their dexterity. We sought to create something that was packaged in manageable moments and that, in its ultimate realization, would answer EA's ad of 1983: "Can a Computer Make You Cry?"

Browsing the Internet has become the second most popular entertainment activity in the country, rivaling television for the attention of America. We wanted to produce something worthy of that attention, something that could compete with television in its storytelling but would not compromise interactivity in the process, and we wanted to create entertainment designed from the ground up specifically for this medium.

It became affectionately known as "The New, New, New Thing":
- New because it was for a new medium with new users.
- New because no game had been episodic before.
- New because it was being distributed by EA.com, a new company.

MAJESTIC started officially in May 1999, when Neil relocated to the Bay Area from Austin, Tex., where he had been the general manager of Origin Systems. Neil had a vision for online entertainment, and EA agreed to provide the seed money and organizational autonomy that was necessary in order to pursue the project. Like the project, the team had to be different than existing teams at EA. It truly had to understand game-making, but it also needed to understand storytelling, sharing disci-

– EXPERIENCES DESIGNING AN EPISODIC INTERNET-BASED GAME

ABOVE. Rooted in the conspiracy theory MAJESTIC sends players to various web links for clues and information. Pictured here are images from just a few of those web sites.

plines with the TV and film business, and how to leverage the best Internet technologies.

The core group that we ultimately assembled had a varied background: Rich Moore had a deep understanding of games from his many years in Atari's U.S. Coin-Op Division and at Capcom, and of the Internet from his time at MediaShower; Gail Oda, our production supervisor, was a seasoned veteran of visual effects production, having most recently worked on *Mission: Impossible 2*; John Danza, who became the creative director, was a feature-film screenwriter with experience directing commercials and music videos; Ralph Guggenheim was one of the founding members of Pixar and producer of the movie *Toy Story*.

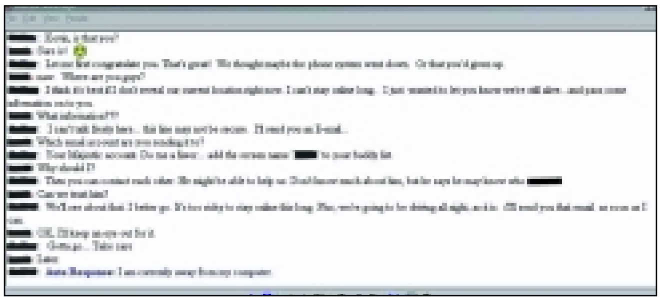
As the team came into place, and with EA's continued support for our vision for online entertainment, MAJESTIC's creative shape took form. While unique in many ways, there are three key features which differentiate MAJESTIC from traditional titles.

It comes to you. The relationship that you have with a game today is like any other entertainment in that you set aside time in your day to interact with it. But unlike other forms of entertainment, it frequently occupies much more of your time than you planned. Ultimately, this is a barrier for games to gain mass-market appeal to adults. As you age, the demands on your time shift: you get a job, find a partner, and perhaps even have children. Finding six hours a night to play DIABLO 2 becomes difficult, if not impossible. In MAJESTIC, we wanted to fundamentally change the relationship that players have with the game. Instead of requiring them to interact for multiple hours at a time to be successful, we wanted to package the experience in much smaller segments.

To achieve this, we leveraged people's relationship with the Internet. The entertainment seeks you out, and at the very least will remind you that you need to play, or better still engage you at that very moment. You could be sitting at your desk and suddenly receive an instant message from a character, you could be driving home when your cell phone rings with a frantic call from a woman who needs your help, or you could be checking your mailbox at work only to find a fax of secret government documents between your copies of *Game Developer*, *PC Gamer*, and *CGW*. By pushing the game toward the player, we're fundamentally changing the dynamic of the experience. This of course creates its own set of unique challenges.

It's played in real time. If a character in MAJESTIC says they'll get back to you tomorrow, they will literally call you tomorrow. This pace of gameplay is very different from most games, which operate at an accelerated pace. What's so powerful about this concept is that once we've established that the game can connect with you through devices such as instant messenging or the phone, anytime an IM window pops up or the phone rings it could be from the game. This adds to and builds the sense of anticipation that the user feels, which for MAJESTIC has become a key creative element.

It's episodic. There was not only an opportunity for us to use the Internet to change the entertainment experience, but also an opportunity to change the development and distribution process radically. By releasing the product episodically, we are able to deliver the initial release of the product to market more quickly and provide a constantly evolving experience. When MAJESTIC goes live, we will have the pilot and two full episodes completed. Moving forward, our internal team will be continually developing future episodes, so we'll constantly be ahead of the players.


[redacted]: Kevin, is that you?
[redacted]: Sure is!
[redacted]: Let me first congratulate you. That's great! We thought maybe the phone system went down. Or that you'd given up.
[redacted]: sure. Where are you guys?
[redacted]: I think it's best if I don't reveal our current location right now. I can't stay online long. I just wanted to let you know we're still alive...and pass some information on to you.
[redacted]: What information??
[redacted]: I can't talk freely here... this line may not be secure. I'll send you an E-mail...
[redacted]: Which email account are you sending it to?
[redacted]: Your Majestic account. Do me a favor... add the screen name [redacted] to your buddy list.
[redacted]: Why should I?
[redacted]: Then you can contact each other. He might be able to help us. I don't know much about him, but he says he may know who [redacted]
[redacted]: Can we trust him?
[redacted]: We'll see about that. I better go. It's too risky to stay online this long. Plus, we're going to be driving all night, and [illegible]. I'll send you that email as soon as I can.
[redacted]: OK, I'll keep an eye out for it.
[redacted]: Gotta go... Take care
[redacted]: Later
[redacted]: Auto Response: I am currently away from my computer.

Via AOL Instant Messenger, players are engaged by MAJESTIC and given hints and information to progress through the game.

Creating the Technology

Building an episodic experience leads you to one conclusion very quickly; you have to decouple technology from content. Technology is the single biggest contributor to schedule unpredictability, and predictability is vital if we want to develop new episodes while users are still engaged in the earlier ones.

Thus, the concept of the platform — a single engine that we develop multiple episodes on top of — was born. Whereas content will evolve monthly, the technology is able to evolve quarterly, and always from a stable foundation. Infinitely and inexpensively scalable, it will provide us with a base that we can progressively expand and leverage over time.

MAJESTIC has unique technology requirements which need to meet a number of key objectives:

- The Internet is our sole medium.
- Integrate a wide variety of communication methods.
- Support a player base exceeding one million users.
- Provide a robust, scalable, cost-effective platform supporting episodic delivery of content and regular feature-set enhancements.
- Create toolsets to enable content creation within the episodic schedule requirements.

Due to our requirements, our definition of what constitutes an engine is broader than the image that the term conjures of a pure technological base and toolset. Indeed our platform has these, but it also includes the capability to leverage existing third-party technologies, and also includes new production processes and pipelines.

In order to achieve all of this, we have developed technology which we call the Experience Server, and we've established a number of key relationships with technology providers. We've put a special focus organizationally on the production process and pipeline, commissioning a "Red Book" that forces us to detail our process. We've hired people who are specifically responsible for the management and deployment of assets against that pipeline, as we've invented the myriad new production processes which will allow us to build the product.

This platform is now at the heart of everything that we do. These technologies, processes, and pipelines provide the key building blocks that enable MAJESTIC and the class of product that it represents.

Infrastructure Overview

As mentioned earlier, we quickly determined that if we were to achieve the objective of predictable episodic delivery, we needed to decouple technology from content. Our technical team was responsible for creating the underlying technological foundation that would deliver assets consistent with the entertainment experience to players (see Figure 1).

Majestic intends to support a player base in excess of one million players and provide an experience that delivers new episodes on a regular basis. Meeting both of these criteria requires an enterprise-class technology platform utilizing a component-oriented software implementation. This fact strongly encouraged and rewarded utilizing topography and server components similar to what EA.com uses for their online service.

Somewhat independently, Synthetic and EA.com both came to the conclusion that server-side Java operating on WebLogic application servers would provide the proper environment for our software development. We selected Netscape Enterprise Servers for their strength in scalability and ease of Java integration, and Oracle 8i for our database system due to its robustness and ability to handle a scalable player base.

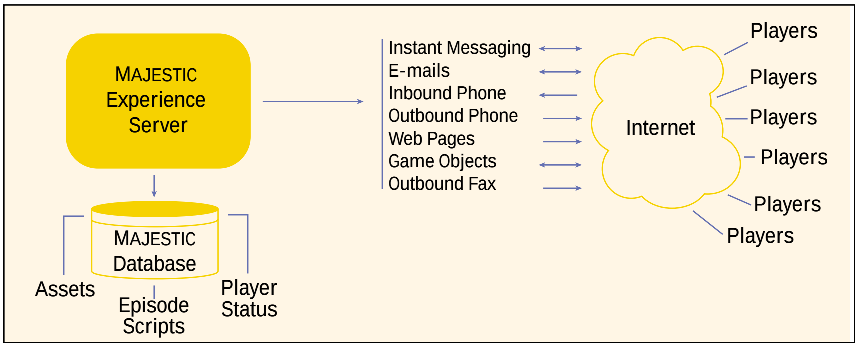
The Java language provides a rich set of core network classes and methods and strongly encourages the reuse of software components. The core classes of server-side Java have allowed us to prototype many of our features, iterate improvements, and provide a working sample that defines a final, production-worthy implementation. Much of the development process behind Majestic has been driven by rapid prototype development, followed by a cycle of iteration, testing, and enhancement. In parallel with this ongoing development, a process of system design, documentation, and implementation was conducted for the final production version of the platform. In essence, this met the needs of product demonstration during development while providing bottom-up testing and validation of the top-down system architecture design process.

The foundation to the Majestic platform is a rich set of Enterprise JavaBeans that are referenced by Java Servlets and JavaServer Pages. This approach encapsulates player-specific data access and separates it from the presentation layer. The separation of the logic for platform functionality, data encapsulation, and final presentation formatting provides us with a seamless means to optimize each portion of our software pipeline. It also allows us to change each portion independently as needed to improve efficiency and include new features and enhancements.

The technical team took advantage of software design patterns in class implementations. One example is the use of a Factory pattern for the asset classes in Majestic. Consider assets as items that are added to a player's inventory as the episode is played. An obvious benefit to this approach is that we can easily extend the asset types supported in our platform by adding additional assetTypes. Existing methods and classes naturally extend to support new assetTypes and maintain current functionality. Some of these assetTypes are functions of our Experience Server and others are derived from third-party technologies which are integrated into the platform.

One of our key development principles was to use technology partnerships to provide several of the delivery systems for the product. We were fortunate that the business arrangement between EA.com and AOL provided us with a unique level of access to AOL's Instant Messaging network and its technology components. Partnerships have also been established with Sentica, to support voice and fax delivery; HotVoice, to provide an Internet-based Unified Messaging System (UMS) for our e-mail, v-mail, and e-fax; EGain, whose eGain Assistant provides the base technology for the conversational AI; Akamai, for hosting of streaming video and audio; and AOL SpinAmp, for streaming audio clients.

By architecting with third-party partnerships in mind we created an easily extendable platform which allows us to replace, add, or remove asset delivery mechanisms over time for Majestic or future products.



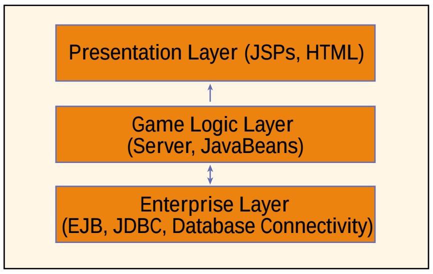

Presentation Layer (JSPs, HTML)
Game Logic Layer (Server, JavaBeans)
Enterprise Layer (EJB, JDBC, Database Connectivity)

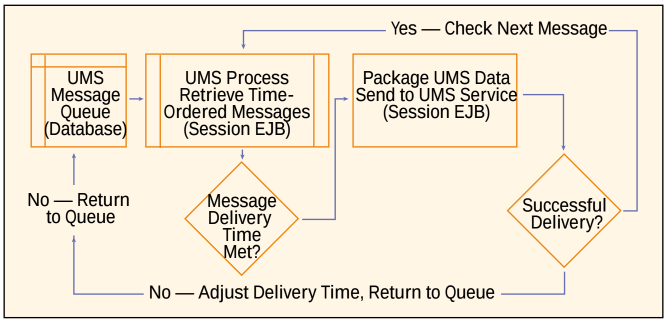
FIGURE 1 (top). Majestic architecture diagram.
FIGURE 2 (bottom). Software layers for Majestic.

Key Challenges

As with all games, the technical development team faced a number of key challenges. Principal on the list were: supporting rapid prototyping while implementing a production-worthy platform, integrating a wide array of communication methods and technologies, and supporting ongoing content development and testing during development of the technology platform. We'll discuss each of these in turn.

As mentioned in the previous section, we conducted parallel development of our enterprise-class software components simultaneously with development of prototypes and early platform versions which used less scalable architectures. As an example, early platform versions combined the presentation logic and game logic. Any change to the visual presentation, human interface, or game logic would require making changes to this component. The segmentation of these software layers is illustrated in Figure 2.

The integration of third-party technologies required establishing


Yes — Check Next Message
UMS Message Queue (Database)
UMS Process Retrieve Time-Ordered Messages (Session EJB)
Package UMS Data Send to UMS Service (Session EJB)
No — Return to Queue
Message Delivery Time Met?
Successful Delivery?
No — Adjust Delivery Time, Return to Queue

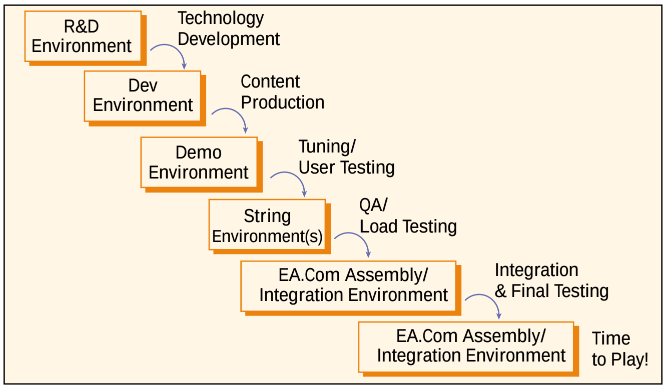


FIGURE 3 (top). UMS processing cycle.
FIGURE 4 (bottom). MAJESTIC environment migration.

clear functional requirements and integrating versions of these products early in the development cycle. As part of our evaluation process, we gave high importance to the developer's ability to meet our player-base growth objectives and the quality and completeness of their product at the time of evaluation. Other business considerations were of course involved in the selection process, yet the challenge and innovation we were striving for helped to energize the work of our partners as it did our internal development staff. Additional complexity was added with the need to ensure that upgrades of our partners' products were synchronized with our development process, so as to minimize the impact on our development schedules. A solution to this third-party integration issue was to isolate data passing via a queue. For example, the Unified Messaging System requires delivery of messages to players at specific times. We utilized a combination of a database queue, a sending UMS process, and the packaged UMS method supplied by our technology partner. We could thus easily upgrade just our technology partner's component and only "touch" our UMS process component to incorporate the changes. This processing cycle is illustrated in Figure 3.

The parallel development of content with technology creates interesting trade-offs and considerations. The need to demonstrate and evaluate the product during development at minimum influences, and more often determines, task priorities for technology development. On MAJESTIC, we completed an early proof-of-concept to demonstrate the potential of the product. This early prototype utilized none of our final code base, but was valuable in driving the definition of the technology platform and blueprinting the functional decomposition of the architecture. This prototype process also served as a model for the leapfrogging of platform versions that occurred during technology development.

One main requirement we determined early on was the need to support several simultaneous server environments at once (see Figure 4). This adds workload to maintain and support each environment, but helps isolate the work of technology development from content creation and testing. Software development proceeded on a specific development environment, implementing new functionality and tests with sample content. Once this new functionality was completed and fully tested, it would migrate over to the content production environment. This isolated many of the possible conflicts between content production and technology development and also assisted in our rapid prototyping approach.

Majestic vs. Traditional Game Development

There are two ways to compare the development of MAJESTIC to traditional game development. One is to compare it with traditional shrink-wrapped products, and the other is to compare it with other online products. The essence of the Internet is the ability of the player to easily combine a number of different activities at once. At times we may be engaged in e-mail, web browsing, or instant messaging. We could have streaming audio or video playing simultaneously. A challenge for MAJESTIC was to provide this range of options to the designer while presenting them functionally integrated within the experience.

An immediate difference for MAJESTIC, then, was the need to provide structure around distinct Internet activities. We solved this by using a variety of technologies and approaches to recognize Internet activities accomplished or discovered by the player that in turn enable other activities or enrich options available to the player. In MAJESTIC, the database stores the content presented to the player as key events are accomplished and game puzzles solved. Unlike traditional games, MAJESTIC does not completely dictate or control the environment of the game — there is no two- or three-dimensional graphical world in which the player operates.

In contrast to most online games, the game experience does not revolve around the requirement to maintain a certain display frame rate or minimize latency. The challenges for MAJESTIC are to optimize performance so that content can be accessed and presented to players quickly over a broad range of bandwidth speeds, and to make the players' accomplishments feel interactive and responsive.

The design decisions made during our development have centered around the need to provide players with personalized content reflecting their "position" in the game — and being efficient about presenting new content and activities as players accomplish key objectives. The episodic nature of MAJESTIC requires that there be a separation between technology and content. This separation supports development by allowing content changes to occur at any time, and provides the foundation for delivery of new episodes as each subsequent one is completed. This principle of encapsulation proved a useful approach toward defining the organization of the technology platform and in defining the structure and organization of data in the database to provide an easy means to publish new episodes.

Why Neil Young Is Like Bill Budge

In 1983, Bill Budge was responsible for shipping EA's first Hall of Fame game, PINBALL CONSTRUCTION SET. Bill did the game design, programming, art, music, manual, and even stuffed floppy disks into baggies for shipment. In 2001, Neil Young is responsible for launching EA's first Internet adventure game, MAJESTIC. Neil did not do the detail of the game design, programming, art, music, manual, or stuff any damn baggies.

But like Bill Budge a generation prior, Neil grabbed a big new game design idea and wrestled it to the ground. Where Bill had been inspired by Apple's work on interface design, Neil was inspired by innovative media like Orson Welles' *War of the Worlds*, everyday communication devices, and movie serials. Both of these guys began with boredom with the game industry's current offerings, but had the confidence to try to find a new way to use our medium.

I watched Neil wrestle with the big new design ideas of MAJESTIC: how to capture the mystery of *X-Files*; how to blur reality and fiction as *War of the Worlds* had done; how to avoid standard CD game formats; how to create and release episodic stories. And then, as Budge had done, Neil made the appropriate organizational decisions for his time. Where Bill had done it all himself, Neil hired experts like John and Rich and Ralph to flesh out the big idea.

Neil has a copy above his computer of EA's original "Can a Computer Make You Cry?" ad, which set a goal to make "software worthy of the minds that use it." Neil doesn't look like Bill, or wear a studded gauntlet like Bill did, but he has brought the same kind of questing intelligence to game design. As much as game development has changed in two decades, the personalities of creative leaders are very similar.

— Bing Gordon, chief creative officer/co-founder, Electronic Arts

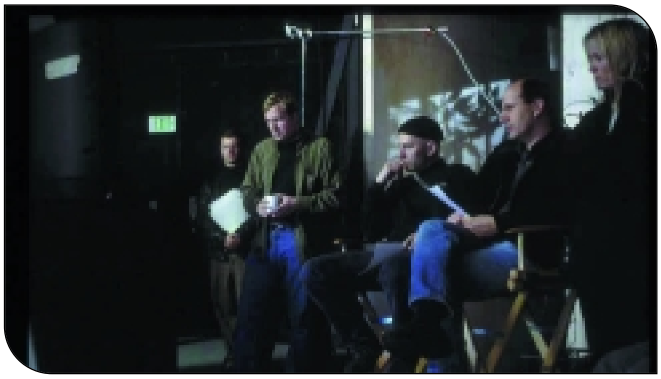
With the exception of the Experience Server, MAJESTIC's discrete technologies are not unique. What makes the experience unique, and what has been our greatest challenge and biggest accomplishment technologically, is how those technologies are used in concert to deliver an entertainment experience that evolves over time, is scalable, is extremely dynamic, and is built to an enterprise-class level.

MAJESTIC Game Design

We've built MAJESTIC and its platform in parallel, and in the process we have learned an amazing amount about how to create games for this medium. While the heart of MAJESTIC's interactivity borrows from adventure games, and the story borrows from episodic television, the surface and pace of the experience are so different that the structure of puzzles and delivery of the story require new and often lateral thinking.

Our key challenge has centered around the pacing of episodes. This meant determining how much content should be delivered in a given day to ensure that users are continually connected with the experience, without requiring them to invest more than 15 to 30 minutes in any given sitting to make progress. Initially, we used just our instinct to determine the right density and volume of content.

Our simple strategy was to drive an instinctive stake in the


Creative director John Danza (in a black cap) and executive producer Ralph Guggenheim (seated to the right) during the filming of a segment for the first episode of MAJESTIC.

ground and then iterate to success, and this rapid prototyping approach has proven to be tremendously valuable. For MAJESTIC, we've focused our energy on building the "pilot," a free demo of the product that users play to get a taste of MAJESTIC before subscribing. It showcases the product and uses every aspect of the platform but is one-quarter to one-third the size of a full episode. By focusing ourselves around the pilot, we found that we had a much more manageable design and implementation problem.

We built the pilot prototype in July 2000 and tested it with a small group of EA employees. Fifteen people participated in that first test, and we were able to get good general feedback. However, it wasn't until our third user test in November 2000 that we really got a good handle on how users were progressing through the game. The knowledge and learning we acquired during that process was primarily enabled by tracking metrics which reported back to us on each user's progress as they advanced through the game. By plotting this information on a graph, we were able to see quickly and visually where users were having difficulty. Our goal from a design standpoint is always to bring as close to 100 percent of the users as possible through to the end of each episode, and never to put people in a situation where they can no longer progress.

If you think about game design, you rarely have an opportunity to gain an accurate and complete picture of how people play a game. In an episodic game it's an essential design and business element to convey as many people as possible through the experience. If we don't, players won't subscribe or continue to subscribe, and if they don't do that then it's all for naught.

What we've learned on MAJESTIC, and specifically how we've tackled and solved design challenges by rapidly prototyping and then measuring real users' interactions with the project, will be a key part of every project that we develop from now on.

New Medium, New Production Processes

Creating an episode of Majestic combines a number of techniques, tools, and tricks that draw from film, television, web, and game production. The successful integration of all these techniques lies at the core of MAJESTIC.

The episodic nature of the game is its greatest production challenge. This rhythm affects every aspect of the experience, from the way episodes are conceived and written, to the design of the technology platform, to the QA process, and even its marketing and publicity. For the moment, let's focus on the nuts and bolts of episodic production.

On any given day, the MAJESTIC crew simultaneously touches three different episodes in the project, each working their way through the production pipeline. In an ideal world, the crew would create only one new episode at a time. In reality, they are producing the current episode, polishing the previous one (based on feedback from QA), and developing story ideas and scripts for the next episode. This constant jumping from past to present to future has a valuable benefit: it compels the team to compare episodes constantly to ensure a comparable level of quality from one to the next.

As described earlier, we recognized early on the need to organize around technology and content separately. The technology team builds the technological components of the platform, and our content team is responsible for the story and game design and development that happens on top of it.

The all-encompassing challenge for the content team is to create an ongoing story with compelling characters that users will want to spend time with and get to know. This is a significant departure from the development of most other games at Electronic Arts. However, with a creative director, executive producer, and others on the team from television, film, and theater, we were able to apply lessons we had learned in those worlds to this interactive medium.

One simple example: the creation of a "story room," a technique used by screenwriters and animation story artists as a war room for story development. It is a place where the project's writing team can work for days at a time while creating a new episode. The room features floor-to-ceiling bulletin boards and whiteboards, and each wall serves a specific purpose. One wall lays out overall ideas for all the episodes in the first season. As the season progresses, the creative team fills the wall with episode information, character descriptions, inspirational artwork, and other material that fill in gaps in the story, but keep us aware of the overall thrust of our seasonal story concept. A second wall is reserved for brainstorming the story structure of a single episode. At a glance we can assure ourselves that the vision for the season on the first wall and the ideas for the current episode on the second wall are consistent. Finally, a third wall graphically deconstructs that episode outline into the specific assets required for production. Once the episode has evolved to suf-

D.W. Griffith and the Language of Film

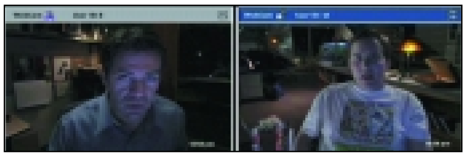
D.W. Griffith is credited for inventing the early language of film, with things like "the establishing shot," "the close-up," "the fade out," and "the cross cut," among other things. I've been trying to understand what the equivalent of "the close-up" or "the cross cut" would be in this new medium we are creating and have a few ideas about that. Our vocabulary is ever growing and we're all slowly becoming fluent. We may even be getting poetic with this new language we've learned.

— John Danza, creative director

ficient detail, it is pitched to the entire team, who give feedback and look for holes in the episode's story logic and gameplay. Only after the story has passed this final test is it memorialized in a script or blueprint document and readied for production.

Another test prior to moving the episode to production is a rough calculation of the number and types of each asset (there are 150 to 200 assets in a typical episode). Based on experience with previous episodes, we attempt to assess whether this asset count fits within the constraints of our production schedule and try to make adjustments accordingly. This can be a difficult task, as the writers have already worked to tell their story as efficiently as possible, but the discipline of working toward a manageable monthly production cycle requires that we be able to predict the workload with a high degree of accuracy.

After a blueprint for the episode has been created, two other events occur in rapid succession: the design of a logic table that captures the triggers and events that comprise the episode, and the creation of a production schedule assigning each asset to the group responsible for it. Delivering an episode of MAJESTIC relies on the production of a variety of assets of different types that must come together to tell the story and entertain the audience. The crew is organized around specific areas of expertise: writing, art and web design, Flash programming, and video and audio production — these being the simplest groupings of the dozen or so types of assets involved in an episode. All the work is performed in-house with the exception of the video and audio assets, which are created by freelance crews under the supervision of our creative director and production management team.



ABOVE. The game will feature streaming video clips such as these from a web cam. RIGHT. Players will be provided links to many web sites, some real and some fictitious — such as this "U.S. Digital Services" site.

While the entire cycle of production for an episode spans eight to ten weeks, the core of the production schedule is typically broken into two segments: a three-week production period to create the various assets for the episode and one week to integrate, test, and polish these assets. Considering the aggressive roll-out strategy of MAJESTIC as an episodic product, it's a very short time schedule, which puts pressure on the production management team to schedule the process in sufficient detail. The entire production process is dependent on the timely creation of assets, all of which must be approved by the creative director to be considered final. We have established a multi-step approval system that allows us to track the status of each asset in our process. This multi-step approval is captured in a customized database to track the progress of the episode. The system generates metrics in a variety of forms to allow our production managers to keep the team focused on meeting the episode's deadline.

While we have limited the number and type of assets in a given episode, we are still learning from the creation of each new episode and applying those lessons to successive ones. We are constantly improving the production pipeline by creating time-saving tools to address each step in our process. Ultimately, every change and enhancement is judged by its ability to enhance the user experience in this uncharted medium of interactive storytelling.

On a creative level, the production of an episode bears a lot of similarity to other production work we have done in film and video. First and foremost, everything must be created to support and enhance the emotional quality of the story and its characters. This is an area that has frequently been overlooked in game design. Where many interactive products are games with a few plot points, we have tried to make MAJESTIC a "story with puzzles." It is our single-minded emphasis on the importance of the story and characters that we hope makes MAJESTIC truly unique in its medium.

Since our production process most closely resembles episodic television production, we have tried to learn as much as we can from that medium. But TV and film can rely on the fact that the technology they use to distribute and present their product (35mm film, videotape, cable TV) are standardized and relatively stable. The intimate relationship between MAJESTIC's technology and its content is typical of computer and online games. Issues of browser and plug-in compatibility complicate our ability to deliver a consistent user experience. Our tech team must carefully integrate these standards, including those of our online partner, AOL. Also, the QA process has a tremendous impact on the production schedule. This is another element that is vastly different from film or TV production, due again to the need for compatibility with a wide variety of online and computer platforms.

MAJESTIC offers the opportunity to explore storytelling in a new and different way. It uses the various communications media of the web to unfold its story. While MAJESTIC appropriates the digital communications techniques we use every day, at times we feel akin to the early directors of movies at the start of the last century — equipped with camera, film, and lights, but trying to learn just how to harness their technology to best tell a story. As film pioneers did, we are able to bring to bear our related expertise from traditional storytelling media, but they only go so far as we undertake this complex project in the new medium of the Internet.

The Future

At the time of this writing, we are in the middle of our fourth user test, with 500 users around the country participating in a closed test of MAJESTIC. Overall, the feedback is great, and people seem to be excited about what we've been able to create.

Although we are constantly trying to make MAJESTIC the best game it can be, we realize that it is ultimately version 1.0 of a new Internet-based entertainment experience. When we look back, I hope that we are able to see MAJESTIC as a commercial and critical success, but regardless it will have provided us with many ideas and insight into leveraging the strengths of the Internet as an entertainment medium.

Toward More Realistic Pathfinding

Pathfinding is a core component of most games today. Characters, animals, and vehicles all move in some goal-directed manner, and the program must be able to identify a good path from an origin to a goal, which both avoids obstacles and is the most efficient way of getting to the destination. The best-known algorithm for achieving this is the A* search (pronounced "A star"), and it is typical for a lead programmer on a project simply to say, "We'll use A* for pathfinding." However, AI programmers have found again and again that the basic A* algorithm can be woefully inadequate for achieving the kind of realistic movement they require in their games.

This article focuses on several techniques for achieving more realistic looking results from pathfinding. Many of the techniques discussed here were used in the development of Activision's upcoming Big Game Hunter 5, which made for startlingly more realistic and visually interesting movement for the various animals in the game. The focal topics presented here include:

Achieving smooth straight-line movement. Figure 1a shows the result of a standard A* search, which produces an unfortunate "zigzag" effect. This article presents postprocessing solutions for smoothing the path, as shown in Figure 1b.

Adding smooth turns. Turning in a curved manner, rather than making abrupt changes of direction, is critical to creating realistic movement. Using some basic trigonometry, we can make turns occur smoothly over a turning radius, as shown in Figure 1c. Programmers typically use the standard A* algorithm and then use one of several hacks or cheats to create a smooth turn. Several of these techniques will be described.

Achieving legal turns. Finally, I will discuss a new formal technique which modifies the A* algorithm so that the turning radius is part of the actual search. This results in guaranteed "legal" turns for the whole path, as shown in Figure 1d.

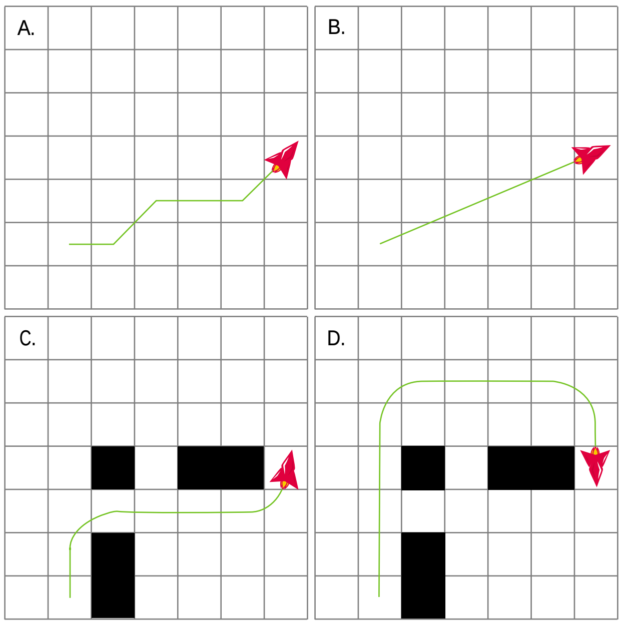


FIGURE 1. Some of the techniques discussed in this article. (A) is the result of a standard A* search, while (B) shows the results of a post-process smoothing operation. (C) shows the application of a turning radius for curved turns. (D) illustrates an A* modification that will enable searches to include curved turns that avoid collisions.

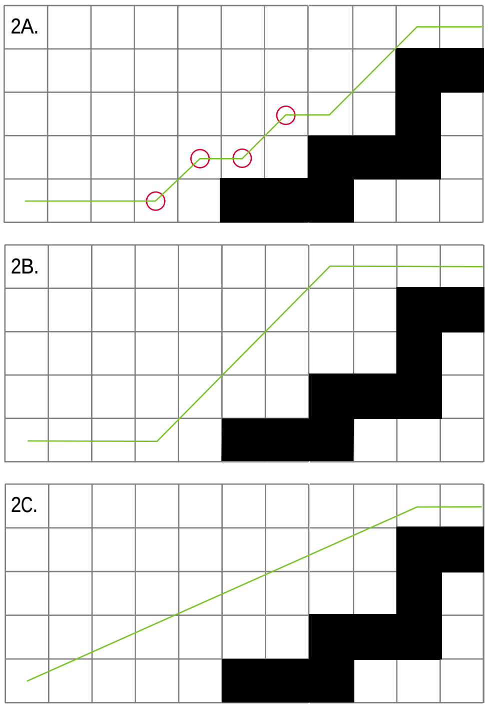
Examining these various techniques will not reveal a single, true "best approach." Rather, which method you choose will depend on the specific nature of your game, its characters, available CPU cycles, and other factors.

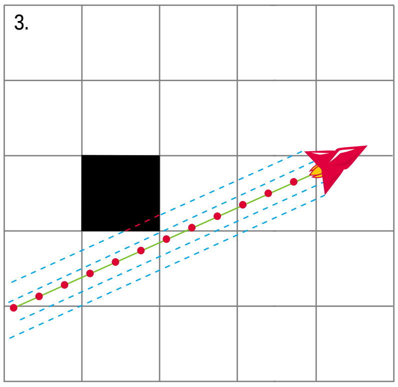
Note that in the world of pathfinding, the term "unit" is used to represent any on-screen mobile element, whether it's a player character, animal, monster, ship, vehicle, infantry unit, and so on. Note also that while the body of this article presents examples based on tile-based searching, most of the techniques presented here are equally applicable to other types of world division, such as convex polygons and 3D navigation meshes.

A Brief Introduction to A*

The A* algorithm is a venerable technique which was originally applied to various mathematical problems and was adapted to pathfinding during the early years of artificial intelligence research. The basic algorithm, when applied to a grid-based pathfinding problem, is as follows: Start at the initial position (node) and place it on the Open list, along with its estimated cost to the destination, which is determined by a heuristic. The heuristic is often just the geometric distance between two nodes. Then perform the following loop while the

MARCO PINTER | *Marco deftly escaped from the rat race in 1998 by selling his development firm to a public company. He currently consults for the game industry as a programmer and designer from his home in Santa Barbara, tinkering with concepts and AI code while watching dolphins swim by. He can be reached at marco@badass.com.*





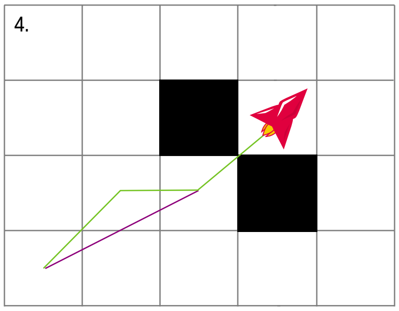


FIGURE 2 (above left). The common zigzag effect of the standard A* algorithm (A); a modification with fewer, but still fairly dramatic, turns (B); and the most direct — and hence desired — route (C). To achieve the path shown in Figure 2c, the four waypoints shown in red in Figure 2a were eliminated. FIGURE 3 (top right). Illustration of the `Walkable()` function which checks for path collisions. FIGURE 4 (bottom right). This smoothing algorithm will leave impossible paths alone.

Open list is nonempty:

- Pop the node off the Open list that has the lowest estimated cost to the destination.
- If the node is the destination, we've successfully finished (quit).
- Examine the node's eight neighboring nodes.
- For each of the nodes which are not blocked, calculate the estimated cost to the goal of the path that goes through that node. (This is the actual cost to reach that node from the origin, plus the heuristic cost to the destination.)
- Push all those nonblocked surrounding nodes onto the Open list, and repeat loop.

In the end, the nodes along the chosen path, including the starting and ending position, are called the waypoints. The A* algorithm is guaranteed to find the best path from the origin to the destination, if one exists. A more detailed introduction to A* is presented in Bryan Stout's article "Smart Moves: Intelligent Pathfinding" (October/November 1996), which is also available on Gamasutra.com (see For More Information).

Smoothing the A* Path

The first and most basic step in making an A* path more realistic is getting rid of the zigzag effect it produces, which you can see in Figure 2a. This effect is caused by the fact that the standard A* algorithm searches the eight tiles surrounding a tile, and then proceeds to the next tile. This is fine in primitive games where units simply hop from tile to tile, but is unacceptable for the smooth movement required in most games today.

One simple method of reducing the number of turns is to make the following modification to the A* algorithm: Add a cost penalty each time a turn is taken. This will favor paths which are the same distance, but take fewer turns, as shown in Figure 2b. Unfortunately, this simplistic solution is not very effective, because all turns are still at 45-degree angles, which causes the movement to continue to look rather unrealistic. In addition, the 45-degree-angle turns often cause paths to be much longer than they have to be. Finally, this solution may add significantly to the time required to perform the A* algorithm.

The actual desired path is that shown in Figure 2c, which takes the most direct route, regardless of the angle. In order to achieve this effect, we introduce a simple smoothing algorithm which takes place after the standard A* algorithm has completed its path. The algorithm makes use of a function `Walkable(pointA, pointB)`, which samples points along a line from point A to point B at a certain granularity (typically we use one-fifth of a tile width), checking at each point whether the unit overlaps any neighboring blocked tile. (Using the width of the unit, it checks the four points in a diamond pattern around the unit's center.) The function returns true if it encounters no blocked tiles and false otherwise. See Figure 3 for an illustration, and Listing 1 for pseudocode.

Since the standard A* algorithm searches the surrounding eight tiles at every node, there are times when it returns a path which is impossible, as shown with the green path in Figure 4. In these cases, the smoothing algorithm presented above will smooth the

LISTING 1. Pseudocode for the simple smoothing algorithm. The smoothing alogorithm simply checks from waypoint to waypoint along the path, trying to eliminate intermediate waypoints when possible.

```
checkPoint = starting point of path
currentPoint = next point in path
while (currentPoint->next != NULL)
   if Walkable(checkPoint, currentPoint->next)
      // Make a straight path between those points:
      temp = currentPoint
      currentPoint = currentPoint->next
      delete temp from the path
   else
      checkPoint = currentPoint
      currentPoint = currentPoint->next
```

portions it can (shown in purple), and leave the "impossible" sections as is.

This simple smoothing algorithm is similar to "line of sight" smoothing, in which all waypoints are progressively skipped until the last one that can be "seen" from the current position. However, the algorithm presented here is more accurate, because it adds collision detection based on the width of the character and also can be used easily in conjunction with the realistic turning methods described in the next section.

Note that the simple smoothing algorithm presented above, like other simple smoothing methods, is less effective with large units and with certain configurations of blocking objects. A more sophisticated smoothing pass will be presented later.

Adding Realistic Turns

The next step is to add realistic curved turns for our units, so that they don't appear to change direction abruptly every time they need to turn. A simple solution involves using a spline to smooth the abrupt corners into turns. While this solves some of the aesthetic concerns, it still results in physically very unrealistic movement for most units. For example, it might change an abrupt cornering of a tank into a tight curve, but the curved turn would still be much tighter than the tank could actually perform.

For a better solution, the first thing we need to know is the turning radius for our unit. Turning radius is a fairly simple concept: if you're in a big parking lot in your car, and turn the wheel to the left as far as it will go and proceed to drive in a circle, the radius of that circle is your turning radius. The turning radius of a Volkswagen Beetle will be substantially smaller than that of a big SUV, and the turning radius of a person will be substantially less than that of a large, lumbering bear.

Let's say you're at some point (origin) and pointed in a certain direction, and you need to get to some other point (destination), as illustrated in Figure 5. The shortest path is found either by turning left as far as you can, going in a circle until you are directly pointed at the destination, and then proceeding forward, or by turning right and doing the same thing.

In Figure 5 the shortest route is clearly the green line at the bottom. This path turns out to be fairly straightforward to calculate due to some geometric relationships, illustrated in Figure 6.

First we calculate the location of point *P*, which is the center of our turning circle, and is always radius *r* away from the starting point. If we are turning right from our initial direction, that means *P* is at an angle of (`initial_direction - 90`) from the origin, so:

```
angleToP = initial_direction - 90
P.x = Origin.x + r * cos(angleToP)
P.y = Origin.y + r * sin(angleToP)
```

Now that we know the location of the center point *P*, we can calculate the distance from *P* to the destination, shown as *h* on the diagram:

```
dx = Destination.x - P.x
dy = Destination.y - P.y
h = sqrt(dx*dx + dy*dy)
```

At this point we also want to check that the destination is not within the circle, because if it were, we could never reach it:

```
if (h < r)
   return false
```

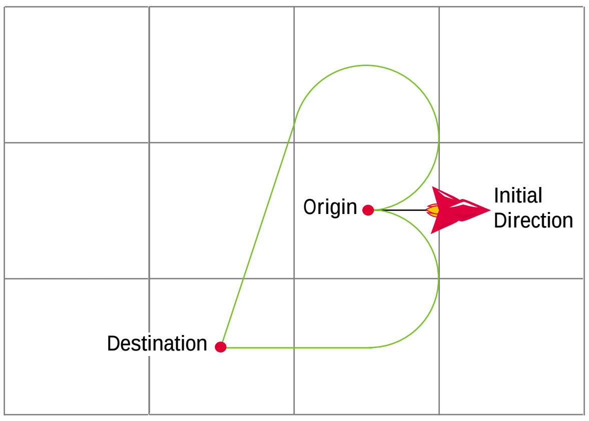
Now we can calculate the length of segment *d*, since we already know the lengths of the other two sides of the right triangle, namely *h* and *r*. We can also determine angle θ from the right-triangle relationship:

```
d = sqrt(h*h - r*r)
θ = arccos(r / h)
```

Finally, to figure out the point *Q* at which to leave the circle and start on the straight line, we need to know the total angle ϕ + θ, and ϕ is easily determined as the angle from *P* to the destination:

```
ϕ = arctan(dy / dx) [offset to the correct quadrant]
Q.x = P.x + r * cos(ϕ+θ)
Q.y = P.y + r * sin(ϕ+θ)
```

The above calculations represent the right-turning path. The left-hand path can be calculated in exactly the same way, except that we add 90 to `initial_direction` for calculating `angleToP`, and later



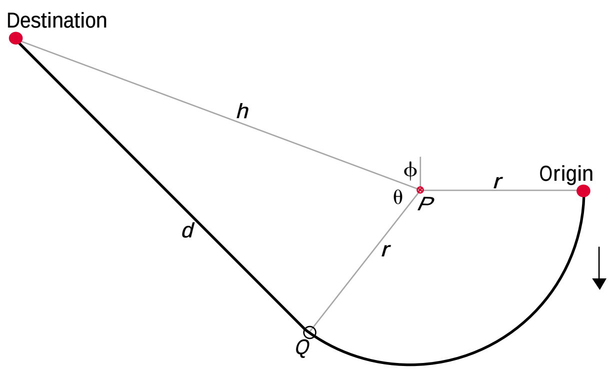


FIGURE 5 (left). Determining the shortest path from the origin to the destination. FIGURE 6 (right). Calculating the length of the path.

we use ϕ – θ instead of ϕ + θ. After calculating both, we simply see which path is shorter and use that one.

In our implementation of this algorithm and the ones that follow, we utilize a data structure which stores up to four distinct "line segments," each one being either straight or curved. For the curved paths described here, there are only two segments used: an arc followed by a straight line. The data structure contains members which specify whether the segment is an arc or a straight line, the length of the segment, and its starting position. If the segment is a straight line, the data structure also specifies the angle; for arcs, it specifies the center of the circle, the starting angle on the circle, and the total radians covered by the arc.

Once we have calculated the curved path necessary to get between two points, we can easily calculate our position and direction at any given instant in time, as shown in Listing 2.

Legal Turns: The Basic Methods

So now that we know how to find and follow an efficient curved line between two points, how do we use this in our pathing? The methods discussed in this section are all postprocessing techniques. In other words, they involve using the standard A* algorithm during initial pathfinding, and then adding curved turns later in some fashion, either in an extended pathfinding or during actual unit movement.

Simple solution: ignoring blocked tiles. We start with the simplest solution. First use the A* algorithm to calculate the path. Then progress from point to point in the path as follows: At any waypoint, a unit has a position, an orientation, and a destination waypoint. Using the algorithm described in the preceding section,

LISTING 2. Calculating the position and orientation at a particular time.

```
distance = unit_speed * elapsed_time
loop i = 0 to 3:
   if (distance < LineSegment[i].length)
      // Unit is somewhere on this line segment
      if LineSegment[i] is an arc
         determine current angle on arc (theta) by adding or
           subtracting (distance / r) to the starting angle
           depending on whether turning to the left or right
         position.x = LineSegment[i].center.x + r*cos(theta)
         position.y = LineSegment[i].center.y + r*sin(theta)
         determine current direction (direction) by adding or
           subtracting 90 to theta, depending on left/right
      else
         position.x = LineSegment[i].start.x
          + distance * cos(LineSegment[i].line_angle)
         position.y = LineSegment[i].start.y
          + distance * sin(LineSegment[i].line_angle)
         direction = theta
      break out of loop
   else
      distance = distance - LineSegment[i].length
```

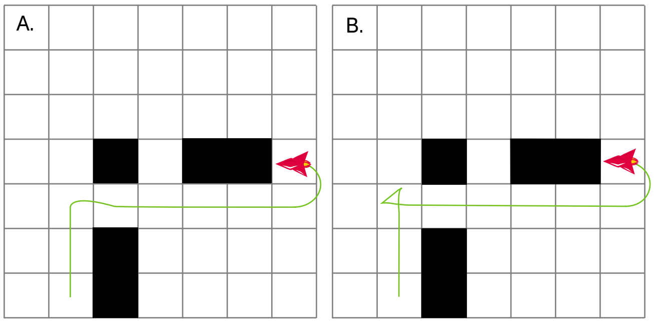


FIGURE 7. Decreasing the turning radius (A), and making a three-point turn (B).

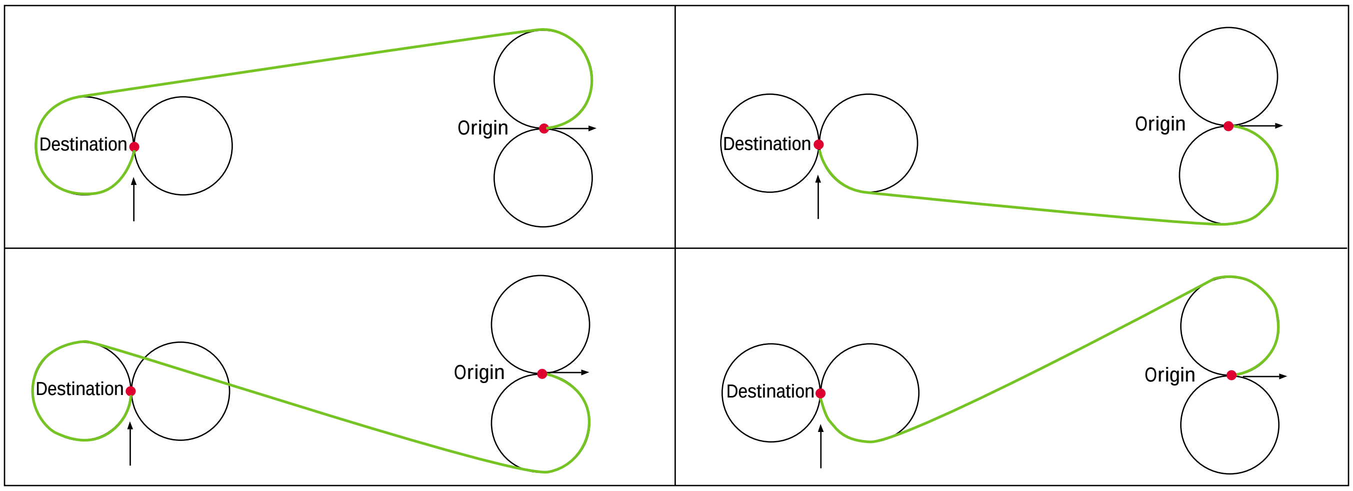
we can calculate the fastest curved path to get from the current waypoint to the next waypoint. We don't care what direction we are facing when we reach the destination waypoint, though that will turn out to be the starting orientation for the following waypoint. If we skim some obstacles along the way, so be it — this is a fast approximation, and we are willing to overlook such things. Figure 1c shows the result of this method. The curves are nice, but on both turns, the side of the ship will overlap a blocking tile.

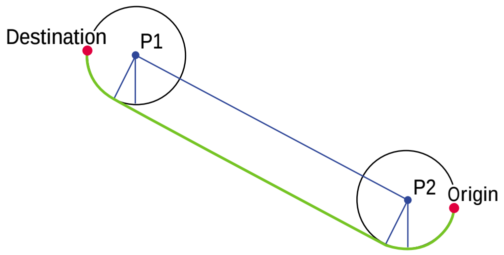
This solution is actually quite acceptable for many games. However, often we don't want to allow any such obviously illegal turns, where the unit overlaps obstacles. The next three methods address this.

Path recalculations. With this method, after the A* has completed, we step through the path, making sure every move from one waypoint to the next is valid. (This can be done as part of a smoothing pass.) If we find a collision, we mark the move as invalid and try the A* path search again. In order to do this, we need to store one byte for every tile. Each bit will correspond to one of the eight tiles accessible from that tile. Then we modify the A* algorithm slightly so that it checks whether a particular move is valid before allowing it. The main problem with this method is that by invalidating certain moves, a valid path approaching the tile from a different direction can be left unfound. Also, in a worst-case scenario, this method could need to recalculate the path many times over.

Making tighter turns. Another solution is that whenever we need to make a turn that would normally cause a collision, we allow our turning radius to decrease until the turn becomes legal. This is illustrated with the first turn in Figure 7a. One proviso is that when we conduct the A* search, we need to search only the surrounding four tiles at every node (as opposed to eight), so we don't end up with impossible situations like the one illustrated in Figure 4. In the case of vehicles, this method may look odd, whereby some lumbering tank suddenly makes an unbelievably tight turn. However, in other cases this may be exactly what you want. Unlike vehicles, which tend to have a constant turning radius, if your units are people, they are able to turn much more tightly if they are creeping along than if they are running. So in order to follow the simple path, you simply need to decelerate the unit as it approaches the turn. This can yield very realistic movement.

Backing up. Our final solution comes from real-world experience. How do we make a very tight turn into a driveway? We back up





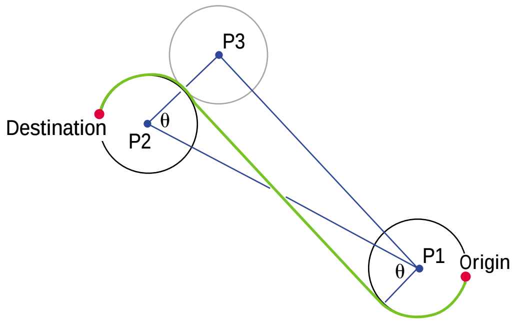


FIGURE 8 (top). Arriving at the destination facing a certain direction. FIGURE 9 (bottom left). Case 1: Traveling around both circles in the same direction. FIGURE 10 (bottom right). Case 2: Traveling around the circles in opposite directions.

and make a three-point turn, of course, as illustrated in Figure 7b. If your units are able to perform such maneuvers, and if this is consistent with their behavior, this is a very viable solution.

Directional Curved Paths

For some of the techniques described later, it is necessary to figure out how to compute the shortest path from origin to destination, taking into account not only starting direction, orientation, and turning radius, but also the ending direction. This algorithm will allow us to compute the shortest legal method of getting from a current position and orientation on the map to the next waypoint, and also to be facing a certain direction upon arriving there.

Earlier we saw how to compute the shortest path given just a starting direction and turning radius. Adding a fixed ending direction makes the process a bit more challenging. There are four possible shortest paths for getting from origin to destination with fixed starting and ending directions. This is illustrated in Figure 8. The main difference between this and Figure 5 is that we approach the destination point by going around an arc of a circle, so that we will end up pointing in the correct direction. Similar to before, we will use trigonometric relationships to figure out the angles and lengths for each segment, except that there are now three segments in total: the first arc, the line in the middle, and the second arc.

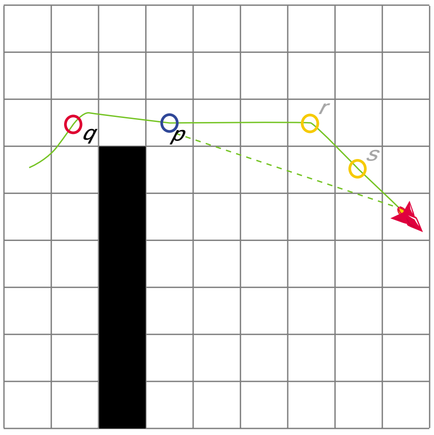
We can easily position the turning circles for both origin and destination in the same way that we did earlier for Figure 6. The challenge is finding the point (and angle) where the path leaves the first circle, and later where it hits the second circle. There are two main cases that we need to consider. First, there is the case where we are traveling around both circles in the same direction, for example clockwise and clockwise (see Figure 9).

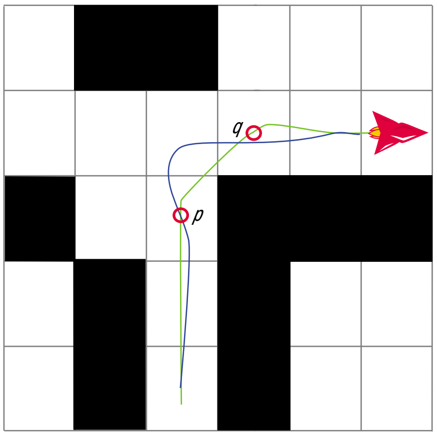
For this case, note the following:

1. The line from *P*1 to *P*2 has the same length and slope as the (green) path line below it.
2. The arc angle at which the line touches the first circle is simply 90 degrees different from the slope of the line.
3. The arc angle at which the line touches the second circle is exactly the same as the arc angle at which it touches the first circle.

The second case, where the path travels around the circles in opposite directions (for example, clockwise around the first and counterclockwise around the second), is somewhat more complicated (see Figure 10). To solve this problem, we imagine a third circle centered at *P*3 which is tangent to the destination circle, and whose angle relative to the destination circle is at right angles with the (green) path line. Now we follow these steps:

1. Observe that we can draw a right triangle between *P*1, *P*2, and *P*3.
2. We know that the length from *P*2 to *P*3 is (2 * radius), and we already know the length from *P*1 to *P*2, so we can calcu-





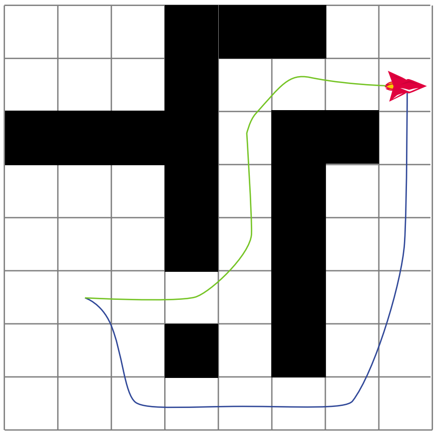


FIGURE 11 (left). One shortcoming of the simple smoothing algorithm. FIGURE 12 (middle). Another shortcoming: the simple smoothing algorithm is unable to find and execute a turn within the legal turning radius. FIGURE 13 (right). The blue line shows the only truly legal path, which the pre-smoothing algorithm can't find, but the Directional search can.

late the angle θ as θ = arccos(2 * radius / Length(*P*1, *P*2))

3. Since we also already know the angle of the line from *P*1 to *P*2, we just add or subtract θ (depending on clockwise or counterclockwise turning) to get the exact angle of the (green) path line. From that we can calculate the arc angle where it leaves the first circle and the arc angle where it touches the second circle.

We now know how to determine all four paths from origin to destination, so given two nodes (and their associated positions and directions), we can calculate the four possible paths and use the one which is the shortest.

Note that we can now use the simple smoothing algorithm presented earlier with curved paths, with just a slight modification to the `Walkable(pointA, pointB)` function. Instead of point-sampling in a straight line between `pointA` and `pointB`, the new `Walkable(pointA, directionA, pointB, directionB)` function samples intermediate points along a valid curve between A and B given the initial and final directions.

Legal Turns: The Directional A* Algorithm

All of the smooth turning methods presented earlier are actually hacks or cheats of one kind or another to achieve the effect of realistic movement. While this is valid and common in game programming, there are times when we may want a formally correct solution that takes into account turning radius and other factors. Comparing Figures 1c and 1d, we see that the only valid solution which takes turning radius into account may require a completely different route from what the basic A* algorithm provides. To solve this problem, a significant modification to the A* algorithm was implemented, which I have termed the Directional A* algorithm.

The basic A* algorithm has two dimensions for every node, namely the X and Y coordinates of the grid. (This can differ if you are using convex polygons or grid meshes, but the concept is the same.) The Directional A* algorithm adds a third dimension, which is the orientation of the unit as one of eight compass directions. This allows the direction a character is moving to be taken into account when determining whether a move from one node to another is valid.

There are several complex issues in the implementation of the Directional A* algorithm which are outside the scope of this article, but can be found in the expanded version of this article on Gamasutra.com. The expanded article incorporates a discussion of the performance of the algorithm, and also introduces hybrid solutions which contain many of the same benefits and run much faster.

A Better Smoothing Pass

The smoothing algorithm given earlier is less than ideal when used by itself. There are two reasons for this. Figure 11 demonstrates the first problem. The algorithm stops at point *q* and looks ahead to see how many nodes it can skip while still conducting a legal move. It makes it to point *r*, but fails to allow a move from *q* to *s* because of the blocker near *q*. Therefore it simply starts again at *r* and skips to the destination. What we'd really like to see is a change of direction at *p*, which cuts diagonally to the final destination, as shown with the dashed line.

The second problem is demonstrated by the green line in Figure 12. The algorithm moves forward linearly, keeping the direction of the ship pointing straight up, and stops at point *p*. Looking ahead to the next point (*q*), it sees that the turning radius makes the turn impossible. The smoothing algorithm then proceeds to "cheat" and simply allow the turn. However, had it approached *p* from a diagonal, it could have made the turn legally as evidenced by the blue line.

To fix these problems, we introduce a new pre-smoothing pass that will be executed after the A* search process, but prior to the simple smoothing algorithm described earlier. This pass is actually a very fast version of the Directional A* algorithm, with the difference that we only allow nodes to move along the path we previously found in the A* search, but we consider the neighbors of any node to be those waypoints which were one, two, or three tiles ahead in the original path. We also modify the cost heuristic to favor the direction of the original path (as opposed to the direction toward the goal). The algorithm will automatically search through various orientations at each waypoint, and various combinations of hopping in two- or three-tile steps, to find the best way to reach the goal.

Because this algorithm sticks to tiles along the previous path, it runs fairly quickly, while also allowing us to gain many of the benefits of a Directional search. For example, it will find the legal blue line path shown in Figure 12. Of course it is not perfect, as it still will not find paths that are only visible to a full Directional search, as seen in Figure 13.

The original search finds the green path, which executes illegal turns. There are no legal ways to perform those turns while still staying on the path. The only way to arrive at the destination legally is via a completely different path, as shown with the blue line. This pre-smoothing algorithm cannot find that path: it can only be found using a true Directional search or its hybrids. So the pre-smoothing algorithm fails under this condition. Under such a failure condition, and especially when the illegal move occurs near the destination, the pre-smoothing algorithm may require far more computation time than we desire, because it will search back through every combination of directional nodes along the entire path. To help alleviate this and improve performance, we add an additional feature such that once the pre-smoothing algorithm has reached any point *p* along the path, if it ever searches back to a point that is six or more points prior to *p* in the path, it will fail automatically.

Other Topics

There are a number of other important related issues which are not discussed here, but are dealt with in the expanded version of this article on Gamasutra.com. They include:

- Speed optimizations for the A* algorithm
- Using hierarchical map divisions to decrease the search space and reduce pathfinding time
- Dealing with smooth movement for prerendered character art, in which there may only be eight or 16 rendered angles for each unit
- Creating realistic movement for turning on roads without having vehicles slide "across the yellow line"
- Reducing the number of path failures
- Timeslicing of long searches
- Incorporating speed and acceleration parameters by either adding speed as an additional dimension to the A* algorithm (the formal method), or other, simpler "cheats"
- Dealing with units that can have a different turning radius depending on speed.

Final Notes

This article has made some simplifying assumptions to help describe the search methods presented. First, all searches shown have been in 2D space. Most games still use 2D searches, since the third dimension is often inaccessible to characters, or may be a slight variation (such as jumping) that would not affect the search. All examples used here have also utilized simple grid partitioning, though many games use more sophisticated 2D world partitioning such as quadtrees or convex polygons. Some games definitely do require a true search of 3D space. This can be accomplished in a fairly straightforward manner by adding height as another dimension to the search, though that typically makes the search space grow impossibly large. More efficient 3D world partitioning techniques exist, such as navigation meshes. Regardless of the partitioning method used, though, the pathfinding and smoothing techniques presented here can be applied with some minor modifications.

The algorithms presented in this article are only partially optimized. They can potentially be sped up further through various techniques. There is the possibility of more and better use of tables, perhaps even eliminating trigonometric functions and replacing them with lookups. Also, the heuristic used in the advanced smoothing pass could potentially be revised to find solutions substantially faster, or at least tweaked for specific games.

Pathfinding is a complex problem which requires further study and refinement. Clearly not all questions are adequately resolved. One critical issue at the moment is performance. I am confident that some readers will create faster implementations of the techniques presented here, and probably faster techniques as well. I look forward to this growth in the field.

FOR MORE INFORMATION

Find an expanded version of this article online at www.gamasutra.com.

GAME DEVELOPER

Pottinger, Dave C. "Coordinated Unit Movement" (January 1999).
www.gamasutra.com/ features/19990122/movement_01.htm

Pottinger, Dave C. "Implementing Coordinated Movement" (February 1999).
www.gamasutra.com/features/19990129/implementing_01.htm

Pottinger, Dave C. "The Future of Game AI" (August 2000).
www.gamasutra.com/features/20001108/laird_01.htm

Stout, W. Bryan. "Smart Moves: Intelligent Pathfinding" (October/November 1996).
www.gamasutra.com/features/19970801/pathfinding.htm

WEB SITES

Steven Woodcock's Game AI Page
www.gameai.com

BOOKS

Game Programming Gems (Charles River Media, 2000)
Refer to chapters 3.3, 3.4, 3.5, and 3.6.

Return

Rogue Entertainment's American McGee's Alice

What do you get when you take a successful independent developer, one of the world's largest game publishers, and a design inspired by a property that is both unique and familiar, Lewis Carroll's *Alice* tales? You get Electronic Arts looking to make a huge splash with their first PC action title, Rogue Entertainment creating AMERICAN MCGEE'S ALICE and a roller coaster ride of success and failure all wrapped up into one little black box with a girl and her enigmatic cat gracing the cover.

AMERICAN MCGEE'S ALICE is a tale of a young girl who is subjected to a tragic incident which leaves her severed from reality and locked away inside the safety of her own mind. Years later, the experience begins to attack Wonderland, turning it into a dark and threatening place, as broken and fragmented as Alice herself. We wanted to create a game that would combine the frantic elements of such legendary games as DOOM and QUAKE with the adventure elements of a TOMB RAIDER. We were shooting for a game that would seamlessly combine those elements with an intriguing story and use the boundless freedom of Wonderland to create fantastic environments in which to present these features. Another goal was to create a strong female heroine devoid of the usual extremely overexaggerated female assets. Alice was to be a character that would cross the usual gender boundaries in action games and bring female gamers into the fold.

Rogue was prepared to carry out these goals through our extensive experience with the QUAKE technology, stemming from five years of working with id Software on mission packs for QUAKE and QUAKE 2, and the N64 port of QUAKE 2. We know the id technology inside and out, which was one of our greatest strengths when we started our discussions with EA. We had a team of nine dedicated professionals, ready to tackle the challenges of Rogue's first full title since our introductory game, STRIFE, four years prior. We also felt that we needed to leave behind the "mission pack" company mantle and that ALICE would let us do just that.

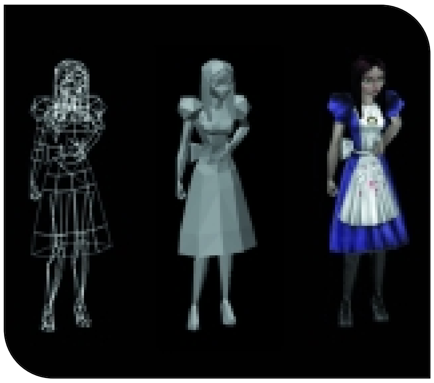
As a company, all of us were enthralled that we would be the ones to tell a new tale of Wonderland and our Alice's adventures through them. The programming team was prepared to enhance and expand the technology to create a platform that would support the immense weight of the content assets. The designers were ready to shape a world so exciting that it would set new standards in level design. The artists were equipped with the ability to transfer the warped representations of a twisted world to the textures and skins of this world. Our modeler/animators wanted to stretch the preconceived notions of Lewis Carroll's characters and add our own additions to the bestiary, with creative flair that would burn these characters into the minds of the players. Needless to say, we wanted ALICE to be the best game that we could make it, and nothing was going to stand in our way of making this happen.

American McGee's ALICE

GAME DATA

PUBLISHER:	ELECTRONIC ARTS
NUMBER OF DEVELOPERS, ARTISTS & CONTRACTORS:	27
BUDGET:	$2.5 MILLION
LENGTH OF DEVELOPMENT:	16 MONTHS
RELEASE DATE:	DECEMBER 5, 2000
PLATFORMS:	WINDOWS 95/98/ME
HARDWARE USED:	DUAL PENTIUM II 500–700'S
SOFTWARE USED:	3DS MAX, CHARACTER STUDIO, TEXTURE WEAPONS, QERADIANT, SOURCESAFE, VISUAL C, MICROSOFT PROJECT, MICROSOFT OFFICE SUITE, PHOTOSHOP, PAINT SHOP PRO, DEEP PAINT 3D
TECHNOLOGIES:	QUAKE 3, FAKK2
LINES OF CODE:	200,000
LINES OF SCRIPT:	30,000
NUMBER OF ART FILES:	5,500
NUMBER OF CUPS OF COFFEE CONSUMED:	I DON'T THINK THAT WE CAN COUNT THAT HIGH

JIM MOLINETS | *Jim laid the foundation for his computer gaming career with concepts for comic books, designs for role-playing and strategy board games, and endless nights of wide-eyed "investigation" of computer games. With over 10 years of professional experience in the industry, creating over 50 levels, and enjoying the role of producer on all four of Rogue's titles, this focus was not wasted. He works his butt off to make sure that Rogue is a success.*



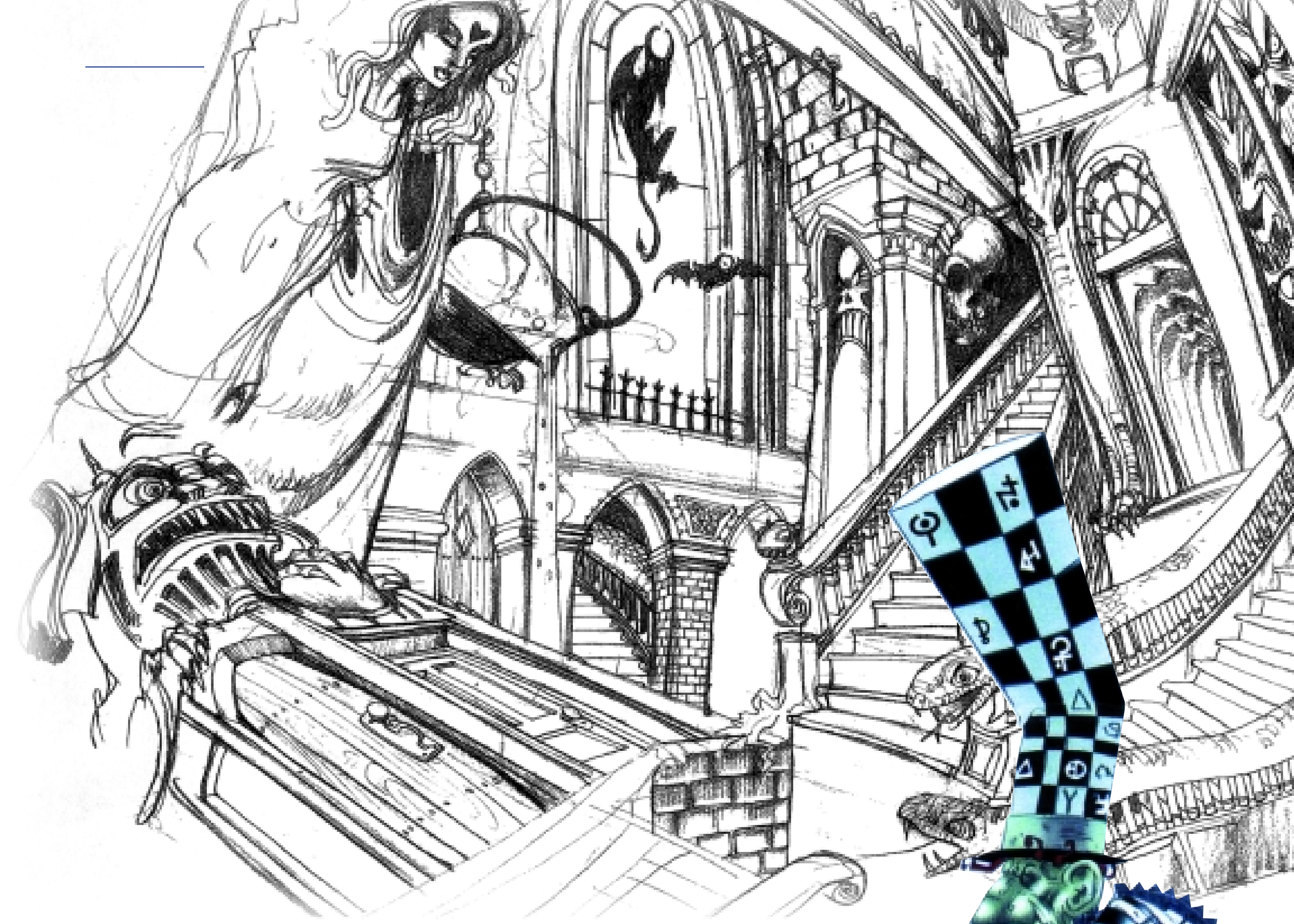
ABOVE. The wireframe, model, and skin for the Alice character. RIGHT. Alice, the Cheshire Cat, and the Mad Hatter in the Fortress. BELOW. Concept sketch of Alice.

What Went Right

1. Company and team growth. While growth presented a challenge to ALICE's scheduling (discussed later in "What Went Wrong"), overall Rogue managed to avoid panicking and hiring warm bodies to fill empty seats. We waited and chose people we felt were right for the company and the team. This was not an easy task, but today we feel that we have one of the strongest, most talented teams in the industry. Just as important as making sure that you make your deadlines is the need to put talented people in place to help you do that. Hiring on a whim has gotten us into trouble before, and our shift to a more refined search paid off during this project.

2. The level of professionalism. This professionalism displayed by the team stretched across all aspects of design to include everyone. There are industry horror stories of teams that are ruined by egomaniacal people. This, for some reason, seems to have become the norm for developers. Rogue has always tried to stay away from that issue, and we feel that the team did just that. During the development of ALICE, people were not interested in who got credit for what, or whose great idea something was, but simply that everyone was working to make the game stronger. This also applied to criticism, another area where egos can get in the way. The ALICE team members openly critiqued each other's work, and this was never viewed as malicious or hurtful, but again as something that would make the product stronger and help everyone.

3. Creative freedom. We never closed the door on creative freedom on any design aspect. I have seen some developers who start with something as a base for an idea, and the team members responsible for the implementation of those ideas are not allowed to deviate from that path. I am not suggesting that people are or should be able to make any change they feel is right, but what worked for us on ALICE was to use

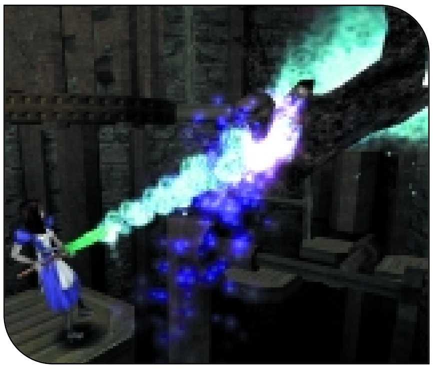


the design as an overall goal of what had to be accomplished with an asset or portion of the game. While the actual work was being done, we encouraged experimentation and creative input so that the entire team could share every aspect of ALICE, not just the individuals responsible for the original ideas. This translated to higher-quality assets and the kinds of inspirational touches that make games really shine.

4. "Guerilla" meetings. Due to the tight time constraints of ALICE, there came a point when team meetings stopped being called; e-mail summaries were sent out instead. After a short period of time, it became obvious that we needed to have meetings regardless of the schedule. To keep the things on track, small "guerilla" meetings were set up where key people met and ironed out any issues on the table quickly. This proved to be a great way to solve major problems without incident. These meetings also served to keep people informed as to what other departments thought might be problems in the future. For the engineers in particular, guerilla meetings became an invaluable tool. They were short meetings, with only the key people from each department, and they stayed very focused, allowing those in attendance to get back to their work quickly.

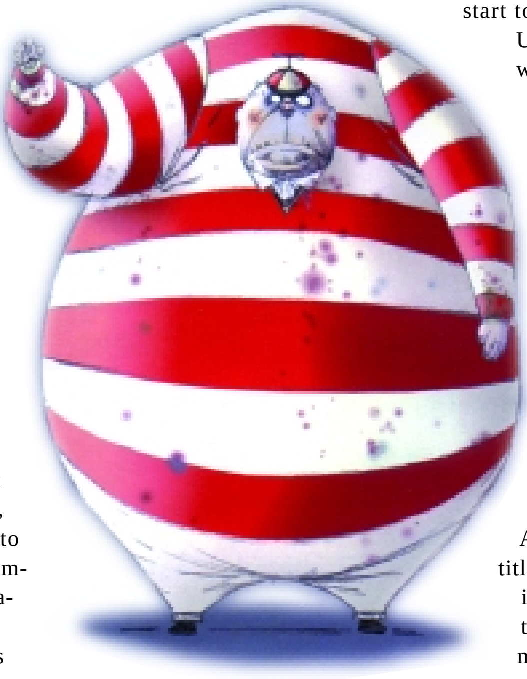
5. The *Alice* intellectual property. I saved this one for last, but most people around the Rogue offices will agree that this was one of the best parts, if not the best part about the project. Most developers look to create something that is new and exciting for their next intellectual property. Sometimes this works out wonderfully (AGE OF EMPIRES, STARCRAFT, DOOM, THE SIMS). Sometimes it does not (BATTLEZONE, INTERSTATE '76). All of those are great games that I have enjoyed playing, but the latter group did not strike the same chord with consumers that the others did. When we first heard that *Alice* was available, we were concerned that this would be something that could go very wrong. Instead, the original body of works inspired the team to try to create something that would do Carroll's works justice. The IP also gave us unfettered

ABOVE. Concept sketch of the Fortress. RIGHT. A rendition of the Mad Hatter.



TOP LEFT. Alice in action in the garden. BOTTOM LEFT. Alice using one of her weapons in the fortress. ABOVE RIGHT. Alice battling the Queen of Hearts' minions. BELOW. A concept sketch for Tweedle Dee.

creative freedom. Who's to say what works and does not work in Wonderland? This is one of the issues with using real-world settings. No matter how fanciful you make the differences between the real world and your game world, people still ground their knowledge in what should work in the real world. The scope of this freedom was upheld by EA, and we let the team tear into it, creatively speaking. This was also something that has been noted by reviewers as one of ALICE's greatest strengths.



What Went Wrong

1. Scheduling. No schedule survives the first milestone intact. There were many factors that forced us to rework the schedule, but the most important was growth. We started the project with nine developers thinking that we could complete a title that, by the end, required about 25. Our first mistake was to think that one modeler/animator could complete the Alice character and all her animations in three weeks. By the end of the project, our young lass had 180 sequences with about 12,000 frames of animation. Some of that didn't make it into the game, but you can tell by those numbers that we were off, way off. Without a doubt, main characters in third-person-perspective games require dedicated artists from start to finish.

Underestimating the amount of time it would take to complete one entire level was another factor that hurt our schedule. In QUAKE 2, it would take one designer about one-and-a-half to two weeks to create and populate a level, if the level was laid out in advance. In ALICE, it took nearly a month. That also did not include the scripting or cinematics, which added at least another week. We were prepared to make Wonderland all it could be, we just didn't have a correct idea as to how long that would take.

These errors, combined with other, similar factors threatened to push ALICE into the "another late software title" category. In order to minimize this impact, we asked the team to go into the infamous Crunch Time about four months before we were supposed to ship

LEFT. Blown away. RIGHT. Checkmate; a chess game ending in Alice's favor.

the product. (I use capital letters because anyone who has been through one of these knows that it simply requires that kind of respect.) That amount of effort burned out everyone, and to no one's surprise, when we needed to ask even more from people at the end, they couldn't push themselves any more than they already were.

2. Communication. A lot of projects credit miscommunication or lack of communication as factors in causing delays in the product development cycle and missing the original ship date. ALICE, unfortunately, was no different. Rogue started as a nine-person team in one office, a war-room atmosphere which fostered the "Rogue attitude" of community and a stream-of-consciousness design flow. That works for a very small group of people, but simply does not work for a team of 25. We knew that moving to new offices during ALICE's design cycle meant changes to the team's organization, allowing for more structure and better information flow. Initially, we tried to leave our style of open office communication in place at the new offices with the expanded team. This did not work as planned, and when coupled with growing to three times the original team size and having an extremely tight timeline, caused a fundamental breakdown in communication. Information that should have been given to all team members about design changes was not disseminated correctly, and sometimes not at all. This led to a "blurred" management structure as we tried to insert people into a new management hierarchy. We also had problems with training new personnel, as the veterans of the company were trying to sort out their new positions, train new people, and keep to the already tight schedule.

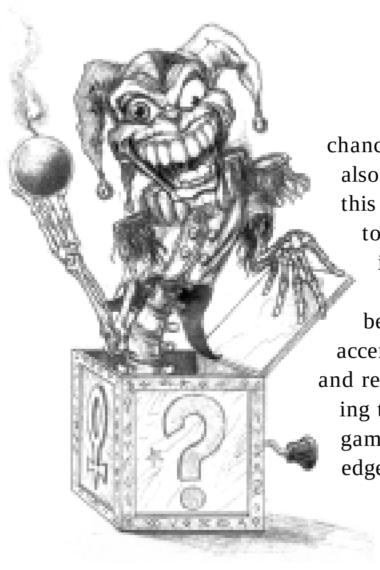
3. Lack of predesign time and assets. There are some in the industry that use the "fly by the seat of our pants" design path. That was something that we were able to use when we were a smaller team, but again, as a company grows, it cannot afford to fall prey to this mistake. With a full team, everyone

ABOVE. The caterpillar.

needs to understand all aspects of the game, and any allowances here only make every other aspect of managing the team and creating the product exponentially harder. We started with a great high concept and some very detailed character information, but when we started to develop the levels and overall structure fully, milestone pressure began to loom, and we had to speed up development to make sure that we would meet our obligations.

While this took care of the short-term issues, it left the team without fully developed goals for the long term. This hurt us most during crunch, when time is of the essence, and several design elements ended up needing to be reworked or tweaked at the last minute. Spending the time to develop the ideas and assets would have saved weeks of work towards the end of the project. Fully developing your ideas before you start asset creation also helps communication flow, because then everyone knows what is supposed to be completed by when and can track the schedule individually. This in turn allows the team to have much more personal flexibility in their asset creation and working schedule.

4. No time allotted to prototype new gameplay innovations. The entire team had one goal in mind when we started this project, to make the best game for the PC in 2000. Most developers have that in mind when they start working on a title, but we felt that by combining powerful technology with our intellectual property and a strong team, there was a very good


The Jack-in-a-Bomb.

chance that we could do this. We also knew that in order to reach this mark, we were going to have to introduce some new and interesting game elements that had not been seen on the PC before, or which had been accentuated for ALICE. Consumers and reviewers alike are always seeking the holy grail of innovative gameplay combined with cutting-edge technology. At the beginning, we knew that the schedule was ambitious. As the development progressed, however, slippage in other areas created an inability to prototype these new innovations fully. We had to fall back on the tried but true action/adventure elements, and if you have been following the reviews of ALICE, you know that this is something that has been seen as an issue with the title. Some of the more interesting elements that were not developed or fully explored were sliding (à la Mario), flying, swimming, and most importantly, more specialty puzzles designed to fit the Wonderland theme.

5. Outside "help." I will be honest here and say that we did have some incredible help on the title by people outside of Rogue. When that worked, it was poetry in motion. However, when it went wrong, it went very wrong. There were instances of work created that had to be redone by internal team members, which pulled them away from more important scheduled items due to the fact that those assets were needed for a more immediate milestone. There was an overall problem getting one particular set of assets from outside that should have been done months before we shipped, but instead they were done by a single contractor with only weeks to go. I am a firm believer in asking for help when you need it, but make sure that the people you are asking to help will really make a difference. If you plan on using them to fulfill milestone requirements, and they miss their milestone or give you assets that cannot be used as is, then you have wasted not only money, but also that more valuable commodity, time.

Lessons from the Trenches

Game development is about taking the good with the bad and the fortitude necessary to realize creative solutions to challenges. It was this credo that kept ALICE chugging along and made its eventual release only one month late. We completed the product on our original budget, built a team, moved during production, and had issues with internal communication, outside contractors, and a lack of predevelopment. If someone were to ask what the single biggest lesson that we took away from the development of ALICE was, without hesitation I would say that it is the need to plan out the entire product in advance, before anyone begins production. Working out the complete game flow and details would have greatly reduced the impact of the problems that arose during ALICE.

Also, as a producer I would tell anyone interested in running a project, no matter what size, to learn how to work in Microsoft Project. It is by no means a magic bullet, but it definitely allowed us to realize very early on exactly where we were going to need assistance and let us start planning our path before it became critical.

The last lesson, and one that we feel the entire team was part of, is that no matter what the problem, there is a solution. It may not be the best or most graceful solution, but it's one that will get the job done. Creativity in this industry is not limited to designing games, but is also an essential part of any successful development process.

The Cheshire Cat.

Return

What Ever Happened to the Designer/Programmer?

Illustration by Claudia Newell

Computer gaming is unique among art forms in that it has undergone a transformation from a solo medium to a collaborative one. For the most part, theater has always been a group effort, and novel writing has always been a solitary activity. However, since the early 1980s commercial computer games have changed from being developed by a single designer/programmer/artist in a room alone with a computer into projects undertaken by large teams in similarly large offices.

This change has had a number of significant effects on game development: management has become much more of an issue; games have become considerably more expensive to develop, limiting the quantity and type of games that get made; the games have changed from representing a distinctly personal vision to that of a group; and the position of lead designer/programmer has been distributed between two separate people. While the first three of these effects may be inherent to the way that computer games have changed as a medium, the last change seems to have come about accidentally, and, to my mind, is not a change for the better.

On one hand, it makes sense from a management perspective that development tasks be divided in the most logical way possible. On cursory inspection it might appear that designing gameplay is an entirely different discipline from actually implementing it. But on the other hand, there are many advantages to including a multi-talented designer/programmer on a team, regardless of the team's size.

A designer who programs will be able to implement the design he or she has in mind perfectly, resulting in time saved both communicating that idea to a programmer as well as reducing the amount of rework required to get that idea working optimally. Furthermore, any programmer knows that coding a game is full of "little" decisions that no amount of designer forethought is going to be able to anticipate, yet it is these programmer choices that ultimately establish the elusive "feel" of a game. All good designers know that — regardless of their own skills — if the programmers working on the game don't have a good sense for gameplay, the final game is not going to be worth a damn. Who better to make sure this "feel" is correct than a programmer who truly understands the game's design?

Designer/programmers have the further advantage of better understanding a game's core technology, leading to thorough exploitation of that technology to create a superior game. Designers with a weak technical background will often fail to understand what can be accomplished trivially and what is nearly impossible to pull off. Indeed, some programmers will use this fact against unsavvy designers, claiming that tasks are impossible merely because they don't want to add them to the game.

It's a sad truth that designers who cannot program are at the mercy of the game's programmers and what they feel like adding to the project. Though a designer/programmer may not add every last feature to the project, if the gameplay is not turning out as hoped, at least a designer/programmer can step into the code and adjust it until it's perfect. Furthermore, designers who can add features to games themselves are much more at liberty to experiment with the gameplay and to try out bizarre ideas that no one thinks will work. In the end, such quirky ideas may turn out to be the best elements of a game.

Despite the many advantages of having a designer/programmer, few companies today seem to use them. When I was looking for a new job two years ago, I found no studios who were interested in hiring someone who would both design and program games. In part this may be because programmers are so rare that those a company does find need to be programming constantly and not spending half their time working on levels or writing design documents. Perhaps there are fears, again from a management perspective, that having a designer/programmer concentrates too much power and responsibility in a single individual.

continued on page 87

continued from page 88

But if one looks at the industry's most revered designers, one will find that a significant number of them started out as programmers — Richard Garriott, Chris Roberts, Steve Meretzky, Jordan Mechner, Dani Bunten Berry, and Tim Schafer, to name just a few. Furthermore, a similarly impressive list continue to program on their projects to this day — Sid Meier, Peter Molyneux, Ed Logg, Brian Reynolds, Jason Jones, and Eugene Jarvis. I was fortunate enough to interview Sid Meier for my book, *Game Design: Theory and Practice*, and questioned him about how he could possibly have time to be lead designer and lead programmer on his projects when so many teams divide that position between two individuals. His immediate answer: "Well, I think they probably spend half their time talking to each other, which is something I don't have to do."

The breakout critical and commercial success of ROLLER COASTER TYCOON is a prime example of the perfect synergy of the designer/programmer. Chris Sawyer was not only the lead designer/programmer on the project, he was the only designer and the only programmer, making the game's development extremely reminiscent of projects from the early 1980s. It seems that Sawyer's filling of both roles gave the game its very personal feel, a unique vision that is a huge part of the game's appeal.

Of course, few commercial games are small enough in technological and gameplay scope to be developed by a single person, but having a designer/programmer on the team can be a boon for any project. Though it's indisputable that many great games have been designed by people who have never programmed, it appears that designer/programmers have a distinct advantage at creating compelling interactive works. Game studios would do well to consider this when assembling their development teams.

RICHARD ROUSE III | *Richard is currently lead designer and occasional programmer on the action/RPG GUNSLINGER at Surreal Software. His past credits include CENTIPEDE 3D, ODYSSEY: THE LEGEND OF NEMESIS, and DAMAGE INCORPORATED. His book,* Game Design: Theory and Practice, *is available from Wordware Publishing, with more information available at www.paranoidproductions.com. He can be reached at rr3@paranoidproductions.com.*