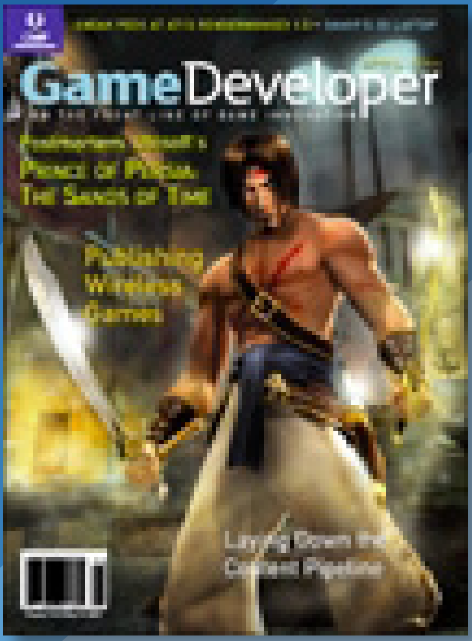
gd

GAME DEVELOPER MAGAZINE

APRIL 2004



GameDeveloper
Postmortem: Ubisoft's Prince of Persia: The Sands of Time
Publishing Wireless Games
Laying Down the Content Pipeline

UBM Tech

GAME PLAN

LETTER FROM THE EDITOR

Player 1 Inserts Coin

Much is made of the idea that success hinges on being in the right place at the right time, and I have to admit I'm starting to believe it. All your careful planning, rigorous training, and heartfelt passion count for just 49 percent of the deal, if that. As for proof, I have strong first-hand anecdotal evidence to support that. There I was, calmly editing this magazine when Jennifer Olsen slapped the baton into my hand, before easing down to more manicured greens. It's been a wild sprint since then, made smoother through the sage guidance of publishing veteran Dominic Milano, and the inspiring support of the rest of the *Game Developer* team. And to top it all off, my lifelong addiction to videogames is finally, finally counting for something (see, Mom!).

Now doesn't that sound familiar? Apart from some eerie parallels between my little shuffle and individual game developers' experiences starting new projects, there's been a sizable amount of systemic musical chairs in the industry as well. Jason Bell and Hajime Satomi were certainly in the right place at the right time, when they took the reins of Turbine and Sega, respectively. Time Warner is inching its way back into the right place, by reestablishing Warner Brothers Interactive. While three or four studios faltered, just as many were reincarnated, such as Level of Detail Software, Team Bondi, and Ready at Dawn Studios, all populated by repositioned industry leaders.

This right place and time business also has ramifications for the types of platforms you create for, within the umbrella of game development. For example, in this issue, Ben Calica takes an extensive look at the wide wireless world, from platforms, to carriers, to developers, to publishers in "The Wireless Gold Rush" (page 28). Through interviews and analysis, he forges order from the chaos, and delivers insider facts and figures, along with informed advice from industry leaders, to help you determine whether the time is right for you to insert your coin.

But things still aren't that easy, as new consoles and handhelds get closer to launch. This year's Game Developers Conference could be one of those right time and place crossroads where we'll learn exactly what's coming up, and how to develop for it. Whether it will make more sense for you to balance your resources in favor of the next generation or continue to create with tried and true SDKs for an installed user base may not be addressed so concretely. Some of those reincarnated studios think they have the answer by focusing exclusively on the next generation—suggesting they've figured out the right place and put themselves there at what they think is the right time. Maybe being that proactive tips the odds in your favor, giving you 51 percent control over your fate. Let's see what happens.

The next issue. There have been some hints dropped by the *Game Developer* staff that we have something exciting planned for the magazine in the near future. Well, the future is rapidly becoming the present, and the next issue marks the magazine's 10th anniversary. What exactly is going on? As the ethereal Dave Bowman says in *2010*, "Something wonderful." While that doesn't really tell you much, what the monoliths actually do to the planet Jupiter and the solar system in that movie is pretty close to what we're doing with *Game Developer*.

So stay tuned.

Jamil

Jamil Moledina
Managing Editor

GameDeveloper
www.gdmag.com
CMP Media, 600 Harrison St., 3rd Fl., San Francisco, CA 94107 t:415.947.6000 f:415.947.6090

EDITORIAL
Editor-in-Chief
Dominic Milano *dmilano@gdmag.com*
Managing Editor
Jamil Moledina *jmoledina@gdmag.com*
Departments Editor
Kenneth Wong *kwong@gdmag.com*
Product Review Editor
Peter Sheerin *psheerin@gdmag.com*
Art Director
Audrey Welch *awelch@gdmag.com*
Contributing Editors
Jonathan Blow *jblow@gdmag.com*
Noah Falstein *nfalstein@gdmag.com*
Steve Theodore *stheodore@gdmag.com*
Advisory Board
Hal Barwood Designer-at-Large
Ellen Guon Beeman Monolith
Andy Gavin Naughty Dog
Joby Otero Luxoflux
Dave Pottinger Ensemble Studios
George Sanger Big Fat Inc.
Harvey Smith Ion Storm
Paul Steed Microsoft

ADVERTISING SALES
Group Associate Publisher
Michele Sweeney e: *msweeney@cmp.com* t: 415.947.6217
Senior Account Manager, Eastern Region and Europe
Afton Thatcher e: *athatcher@cmp.com* t: 404.658-1415
Account Manager, Northern California and Midwest
Susan Kirby e: *skirby@cmp.com* t: 415.947.6226
Account Manager, Western Region and Asia
Craig Perreault e: *cperreault@cmp.com* t: 415.947.6223
Account Manager, Target Pavilion, Education, and Recruitment
Aaron Murawski e: *amurawski@cmp.com* t: 415.947.6227

ADVERTISING PRODUCTION
Advertising Production Coordinator Kevin Chanel
Reprints Julie Rapp e: *jarapp@cmp.com* t: 510.985.1954

GAMA NETWORK MARKETING
Director of Marketing Michele Maguire
Senior Marcom Manager Jennifer McLean
Marketing Coordinator Scott Lyon

CIRCULATION
BPA INTERNATIONAL
Game Developer is BPA approved
Circulation Director Kevin Regan
Circulation Manager Peter Birmingham
Asst. Circulation Manager Lisa Oddo
Circulation Coordinator Jessica Ward

SUBSCRIPTION SERVICES
For information, order questions, and address changes
t: 800.250.2429 or 847.763.59581 f: 847.763.9606
e: *gamedeveloper@halldata.com*

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

EDITORIAL FEEDBACK
editors@gdmag.com

CMP MEDIA MANAGEMENT
President and CEO Gary Marshall
Executive Vice President and CFO John Day
Executive Vice President and COO Steve Weitzner
Executive Vice President, Corporate Sales and Marketing Jeff Patterson
Chief Information Officer Mike Mikos
President, Technology Solutions Robert Faletra
President, CMP Healthcare Media Vicki Masseria
Senior Vice President, Operations Bill Amstutz
Senior Vice President, Human Resources Leah Landro
VP and General Counsel Sandra Grayson
VP, Group Publisher Applied Technologies Philip Chapnick
VP, Group Publisher InformationWeek Media Network Michael Friedenberg
VP, Group Publisher Electronics Paul Miller
VP, Group Publisher Enterprise Architecture Group Fritz Nelson
VP, Group Publisher Software Development Media Peter Westerman
VP & Director of CMP Integrated Marketing Solutions Joseph Braue
Corporate Director, Audience Development Shannon Aronson
Corporate Director, Audience Development Michael Zane
Corporate Director, Publishing Services Marie Myers

CMP United Business Media | CMP Game Group

WWW.CMPGAME.COM

Return

INDUSTRY WATCH

KEEPING AN EYE ON THE GAME BIZ | *kenneth wong*

Origin facing closure. After shutting down Westwood studios and relocating Maxis to Lost Angeles, videogame publishing giant Electronic Arts is reportedly putting Origin Systems on the chopping block. The Texas-based Origin staff was offered a choice between relocation to California or a severance package. Best known for the ULTIMA ONLINE and WING COMMANDER franchises, Origin is a wholly owned subsidiary of EA.

Nokia admits N-Gage not engaging enough buyers. Nokia's CEO Jorma Ollila, who has consistently expressed optimism for the N-Gage in the past, finally admitted to *Financial Times* sales of the device were "the lower quartile of the bracket we had as our goal." Ollila says he will wait until 2005 before passing judgment on the hybrid phone-game device. Initially priced $299, the device is currently selling for $100 less. Except for announcing the shipping of 600,000 units of the device, Nokia has not made public the retail sales figures. The company originally hoped to sell nine million units in the first two years of product launch.

For a verdict on N-Gage, tune in sometime in 2005, says Nokia.

India touts game gods. Following in the footsteps of its neighbors China, South Korea, and Singapore, the Indian government is aggressively marketing its game development talents through the Federation of Indian Chambers of Commerce and Industry. Capitalizing on its reputation as one of the outsourcing Nirvanas, India looks to establish partnerships between international game publishers and the country's leading interactive entertainment firms, such as Dhruva, Crest, and Jadoo. Last year Dhruva signed up to work on a Microsoft Game Studios project due for release in August 2004.

3D mobile gaming contest underway. Organized by Discreet and its mobile game partners (including Nokia, Orange, In-fusio, Nvidia, Criterion, Ideaworks3D, Fathammer, IBM, Intel, and others), the first international 3D mobile gaming competition (www.3dmobilegamingcontest.com) is underway. Launched at the Imagina 2004 tradeshow (Monaco), the contest allows independent mobile gaming artists, developers, and 3D freelancers to carry out their creative concepts for mobile games. It features more than $50,000 worth of prizes, trips, software licenses, co-production opportunities, and a chance to meet with mobile game development luminaries.

Send all industry and product release news to news@gdmag.com.

THE TOOLBOX

DEVELOPMENT SOFTWARE, HARDWARE, AND OTHER STUFF

Alienbrain Studio 7. NXN Software begins shipping Alienbrain Studio 7 on April 15. Performance improvements include multi-threading to take advantage of multiple processor workstations and servers, and faster creation of thumbnail images. And among the 40 new features are change sets for streamlined check-ins and a revised administration module that can use Windows Server databases or LDAP servers to streamline the management and authorization of users while supporting a single sign-on. Alienbrain Studio 7 will be previewed at GDC 2004. www.nxn-software.com

RT/Shader makes shaders easy. RTzen launches a shader development tool, RT/Shader, at GDC. It allows one to create high-level shader programs visually—by assembling visual elements such as textures and lighting effects. The resulting shaders are previewed in real-time for immediate feedback. RT/Shader supports the Direct3D HLSL and Nvidia's Cg shading languages, and includes a code editor in its IDE for viewing and tweaking the code that results from its calculations. The program costs $1,995. www.rtzen.com

3D Mega Capturor DF. InSpeck has announced its latest 3D capture device, the 3D Mega Capturor DF, which can scan a wider-sized range of objects than can its other scanners without requiring swapping of lenses or the use of two scanners. The DF stands for Dual Field, the two cameras it features. This provides the same capability to switch from small to large objects the company's 3D Mega DF product offers, only with greater convenience, making the process of scanning various sized objects or multiple levels of detail much faster. Price is not yet announced. www.inspeck.com

—Peter Sheerin

UPCOMING EVENTS CALENDAR

COMPUTER GAME TECHNOLOGY CONF. 2004
DAYS HOTEL & CONFERENCE CENTRE
Toronto, Canada
April 8–10, 2004
Cost: $220–$250
www.cgt.auc.ca

GAME NETWORKS CONFERENCE & EXPO
HYATT REGENCY HOTEL
Long Beach, Calif.
April 27–28, 2004
Cost: $499–$799
www.scievents.com/gamenet

ATI's RenderMonkey 1.5

by ron fosner

With most major graphics card vendors offering up DirectX9 and OpenGL 1.4 compliant (shader-capable) hardware, real-time shader development is becoming more viable and important. To help further this technology, ATI plans to release version 1.5 of its RenderMonkey shader development utility at GDC 2004. There may be some differences between the version reviewed here, which was a preview version available at press time, and the final 1.5 version.

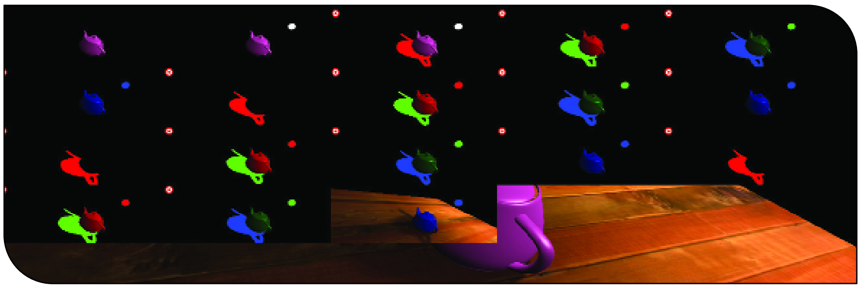


FIGURE 1 . Generating soft shadows typically requires many passes. RenderMonkey 1.5 lets you preview each pass separately for fine-tuning the shader.

RenderMonkey 1.5 is a huge improvement over 1.0. ATI has spent a lot of time listening to the suggestions from developers and artists, adding the features that are most important to them. To say the entire interface has been revised to make it easier to write shaders would be an understatement. Featuring improved editing, better error reporting, support for multiple render targets, and better workflow integration, and an additional shader language, this is a significant update.

Editing features and OpenGL support. The editing environment retains most of the modular design of the previous version but features some drastic revisions in terms of streamlining interaction with the user. The entire interface has been revised to resemble Visual Studio.NET. Many of the windows are now floating windows that can be moved outside the main window or docked, as in Visual Studio. In fact the syntax highlighting and editing preferences are all designed to mimic those found in Visual Studio, since that's the development environment most folks writing PC games are using. The syntax colors are also configurable. Most of these underlying rules for the environment configuration are editable XML or text files, allowing you to easily create custom rules for syntax, project-specific variable declarations, and so on. There are syntax files for DirectX's low-level shader code, DirectX's high-level code (HLSL), and OpenGL 1.4's shader language (GLSL).

Though the HLSL compiler is part of DirectX, it's part of the driver in OpenGL (I'll let you ponder the shading language optimization implications). This means you'll need a card that contains at least an OpenGL 1.4 level driver (which should be available from both ATI and 3DLabs by the time you read this) to try out GLSL. Using a high-level shading language insulates you from the actual assembly language running on the graphics card. However, whenever you need to drop down to the assembly level, RenderMonkey lets you view the low-level code it generates. While you can save or paste the generated file and turn it into a shader, there's no option to do this directly from the user interface.

The shader setup has also been streamlined. RenderMonkey supports intelligent drag-and-drop function, making it much easier to add and reference textures, for example. The creation and editing of shader constants has also been improved.

Rendering environment changes. The new version also makes good use of tooltips—those popup boxes that are common in most development environments. Not only do you get the usual current variable values popping up (even for complex types such as matrices and colors), but you also get fairly intelligent tooltips for other things: file paths for model files, compilation targets for shaders, and (my favorite) the description element. Normally this is just a chunk of text describing the current workspace or effect. However when your mouse hovers over it, the tooltip reveals the entire description, making it easy to see the works in progress or to-do items.

The rendering window now has hot-key support for things like forcing texture or models to be reloaded, toggling the bounding box for objects, saving the current image, or, most useful, displaying the intermediate rendering pass results. For example, creating a soft shadow around an object typically requires blending many passes to get a soft outline.

When rendering something as complex as this (14 shadow and three object passes), it's a big help to see the results of each pass. With RenderMonkey, all you need to do is press the P key (for passes) and you can see the results of each intermediate pass (Figure 1), where each intermediate pass is shown tiled in the main rendering window.

Ron Fosner | *Ron is a 3D programmer and consultant, and author of* Real-Time Shader Programming *(Morgan Kaufmann). Reach him at rfosner@gdmag.com.*

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

Another new smart feature is the ability to use the mouse buttons and mouse position as inputs to the shader. This makes it easy to trigger different rendering paths when testing out alternate algorithms and when you need to switch quickly between different paths or to lerp between values as a function of mouse position.

In addition to standard variables such as Projection Matrix, ViewProjection Matrix, and WorldViewProjection Matrix, version 1.5 adds many new predefined variables, including viewing parameters, many more cycling parameters (plus the ability to set the cycle time for these variables), some random values, and the current rendering pass index.

RENDERMONKEY 1.5
✶✶✶✶

STATS
ATI Technologies, Inc.
33 Commerce Valley Drive East
Thornhill, ON
L3T 7N6 Canada
http://www.ati.com/developer/tools.html

PRICE
Free

SYSTEM REQUIREMENTS
Windows XP, 2000 SP2, ME, or 98 SE, DirectX 9.0b or OpenGL 1.4 installed DirectX 9.0/OpenGL 1.4-compliant graphics card, 128 MB RAM, 180 MB free hard drive space.

PROS
1. The only OpenGL shader tool available.
2. Indifferent to GPU chipset—all you need is DX9 or OpenGL 1.4 support.
3. Can be integrated into production pipeline with some customization effort.

CONS
1. No Intellisense or Autocompletion.
2. Really requires a DX9 and/or OpenGL 1.4 capable graphics card. The DX hardware emulation driver ("REF") is excruciatingly slow.
3. No debugging of shaders.

Workflow improvements. Unless you're working in a large development group with someone devoted to tools development, you've really had no way of shortening the shader development cycle other than to edit the shader code, fire up your rendering engine, load in the new code, test the code, then reedit the shader all over (lather, rinse, repeat). ATI's RenderMonkey 1.0 (for HLSL) and Nvidia's CgFX Viewer (for Cg) were really the only prototyping tools out there, and neither really fit into a production environment. With RenderMonkey 1.5, ATI has hit most of the usability improvements that will let you speed up shader editing and testing.

RenderMonkey has added support for FX files through an improved FX file exporter. Whenever possible, it will convert to standard FX file notation. If there's no FX-file equivalent then it will stick a comment in the top of the FX file. Additionally there's now built-in support for .X mesh files.

You can enable auto-refresh for model and image files as well as specify external editors for these file types so you can launch editors from inside the RenderMonkey IDE. Thus if you discover a texture or a model that needs tweaking, you can have RenderMonkey launch Photoshop or 3DS Max (or whatever tools you specify), edit the file, save it, and switch back to RenderMonkey. RenderMonkey can be set to check for updates on the current shader resource files and will automatically reload them if it detects they have been edited.

ATI has also released the RenderMonkey SDK, which allows you to integrate it with your own development tools. Using the SDK you can write your own plug-ins for RenderMonkey, to create a loader for an unsupported mesh format, to make an exporter for your own unique file format, or even a dynamic shader updating mechanism hooked into your own game engine.

So who should get this monkey? Short of writing your own shader editor interface for your game engine, the latest version of RenderMonkey from ATI comes as close as you can get. In fact, with the SDK and some effort on your part, you can probably achieve that as well. ATI has made some significant improvements to the RenderMonkey interface as well as streamlined the editing effort. However, there is still some room for improvement. The rendering window can't be dragged outside the main window or made fullscreen (say, onto a second monitor), the lack of Intellisense and Autocompletion in the shader editors is surprising, and it would be nice if there were more mesh formats supported natively other than just .3DS and .X. The biggest omission is the missing direct integration with 3DS Max and Maya. RenderMonkey still has an "artist mode," where only certain values (such as colors) can be changed in an effect—allowing you to safely hand off shaders to non-programmer artists for tweaking.

Despite these shortcomings, this is the first tool specifically designed to streamline the shader writer's efforts and it's an excellent tool for that purpose. If you write shaders but haven't checked out RenderMonkey yet, then try out version 1.5.

Sharp's Actius RD3D

peter sheerin

Stereoscopic computer displays using various techniques have been around for many years. But they have never quite caught on, due to the technology's inherent limitations. Sharp's new technology, however, overcomes several of these issues and stands a good chance of turning the tide and bringing realistic stereo imagery to far more applications.

The Actius RD3D features an autostereoscopic display—one that provides a stereoscopic image without the need for wearing glasses or viewing tricks (such as crossing your eyes) to create a true 3D image. In the past, displays have accomplished this with various filters that can provide one or several different viewing areas, but such an approach renders the display useless for normal work, since the filters obstruct or distort non-stereo

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

images and application windows.

On a desktop, this is not so big a problem, since it is fairly easy to connect two monitors—one for stereo and one for everything else. Obviously, though, this is not a viable solution for a laptop computer. Sharp's solution uses a liquid-crystal-based vertical-stripe filter that can be disabled: turn it on, position your eyes in the right spot, and you get a convincing stereo image; turn it off and you'd never know there's anything special about this computer.

The $2,999 Actius RD3D is a large, powerful portable computer that will probably spend most of its time plugged into AC power. It features a desktop 2.8GHz Pentium 4 processor, an Nvidia GeForce4 440 Go graphics chip, a 15-inch display, four USB 2.0 ports, a 4-pin FireWire port, one PC CARD slot, and a 5-in-1 flash memory reader. At 11.7 pounds, it'll make you hesitate to lug it around everywhere. But these issues don't detract from what is an excellent development platform to start experiencing stereoscopic display and updating your game design practices and tools to support true 3D display.

Running 3D games. As previous efforts at creating stereoscopic games have relied on liquid-crystal shutter glasses, which resulted in eyestrain because most home monitors are capable of displaying only at low refresh rates, very few games in recent years have been designed with this technology in mind. Performance shortcuts such as placing HUD, cockpit, and map displays in 2D space (instead of in 3D space) don't matter when playing a game on a normal monitor, but they can make playing a game in 3D mode impossible. The display driver makes some assumptions about depth perception and the distance of objects in a scene if the game doesn't provide a built-in stereoscopic mode. This can result in display artifacts and occasional double visions that vary from game to game.

In conjunction with Nvidia, Sharp has posted on its 3D display web site (www.sharp3d.com) a list of games that work in stereo mode on the RD3D, ranking the value and accuracy of the stereo imagery to show how well a particular game will work in stereo. For example, flight simulation games don't fare well because much of the display geometry is focused on infinity. On the other hand, nap-of-the-earth helicopter simulations such as COMANCHE are interesting exceptions.

Sharp's Actius RD3D auto-stereoscopic laptop

In addition to some still-image stereoscopic editing and viewing applications, our test unit came installed with a few games to demonstrate the capabilities of the unit—customers will receive JAMES BOND 007: NIGHTFIRE, TIGER WOODS PGA TOUR 2003, and NEED FOR SPEED: HOT PURSUIT 2. I also installed Madden NFL 2004. In playing these games, I found that the stereo effect varied from okay (QUAKE II) to very good (SPIDER-MAN and MADDEN). After a little bit of experimenting, I was able to find the sweet spot that enabled the full stereo experience. Moving slightly to one side or the other, or tilting your head a bit, first resulted in double images at the edges of the screen. When I moved more, the entire screen became double images. Getting the software and notebook into stereo mode at the same time is no easy task. The stereo effect only works at 1,024x768. If a game defaults to 800x600 and triggers the stereo mode, you must disable stereo in the display driver control panel, reconfigure the game, and enable stereo in the control panel again. That'll be a significant obstacle hindering mass acceptance.

When I was in the zone, the effect was very good—as good as or better than the previous auto-stereoscopic displays I have viewed in the past, and far better than the monochrome anaglyphs of Mars that NASA has been providing us with.

However, some high-contrast areas of the stereoscopic images exhibited slight ghosting. (Don't get me started on NASA's decision to post virtually all of the Mars rovers' stereo images using technology first pioneered in 1858, but this could explain a few things about the Shuttle.) Since the sweet spot is only an inch or two wide, the zone is quite small. So the effect works very well when you're intently concentrating on a game to the point that your posture is static, but it breaks down to various degrees when playing sports or similar games where bobbing or tilting your head is a natural response.

Running 3D tools. No currently shipping DCC software has a real-time shading display that will work in stereo mode with the RD3D, though many can be made to render stereoscopic movies that will work. Sharp says 3DS Max 6 and Lightwave can render two camera views to a properly interlaced AVI movie that will create the proper stereo effect, and that Maya and Softimage can do this indirectly—you'll have to define multiple camera paths and compose the stereo movie in Premiere. This prevents you from gaining the benefits of depth perception while creating models.

For this stereoscopic technology to work for game developers, DCC vendors and game engine makers will have to add direct support for Sharp's vertical interlaced stereo mode (and the other viable stereoscopic formats—page flipping, horizontal interlacing, and so on) to their programs, so developers can generate content and games that will work seamlessly in stereo mode without the artifacts and compromises that occur when the display driver tries to recreate a stereo scene after the fact. For now, getting access to the SDK requires developers to sign an NDA, but this technology has enough potential that you should seriously consider signing up.

★★★★ | Actius RD3D
Sharp Systems of America
www.sharp3d.com

Peter Sheerin is the product review editor for Game Developer.

Return

Levitating Physics

Midway's Brian Eddy on mixing realism with psychic powers

Imagine picking up the phone to hear Havok tell you your game is a brilliant implementation of their physics engine. Midway's Brian Eddy got that call as he was finishing up PSI-OPS: THE MINDGATE CONSPIRACY. The third-person action game gives its commandos psychic powers and the ability to manipulate objects, enemies, and environments using the standard Havok 2 physics package. According to Tom Lassanske, a developer relations engineer at Havok, this illustrates how a creative developer can offer unique gameplay elements to the creative player through physics.

Brian Eddy is RVing you right now.

This kind of validation prior to a game's release can certainly provide a confidence boost, although Eddy has 14 years of experience to rely on to tell him he's on the right track. While serving as lead designer and executive producer of the title represents his first console foray, he cut his teeth at Midway creating arcade games such as ARCTIC THUNDER and pinball titles such as MEDIEVAL MADNESS and ATTACK FROM MARS. Havok's seal of approval notwithstanding, we were curious how Eddy ports over to the console market.

GD: How did you choose to develop a third-person action game?

BE: We wanted to be able to show the main character physically doing the powers and interacting with the world on screen. It's much more satisfying to see your character actually doing the Psi powers, especially since most of them are very unique to this game.

GD: What attracted you to the idea of psychic soldiers?

BE: Most of the major world governments actually have Psi-Ops programs. Some of the powers are enhanced from the real life version for gameplay, but others, like remote viewing (RV) are very similar. In doing a lot of digging into the covert government Psi programs, it was pretty amazing and scary the amount of research that has gone into Psi powers. Most of the information we found was only recently made public or declassified. Tie that in with the fantasy aspect of actually having these extreme powers and being able to use them in a real world setting just sounded like a lot of fun.

GD: To what extent does pinball's play/reward model carry over?

BE: When you do something well on a pinball machine it will reward you with a big light show, sound, music, and even a free game. When you do something well in a videogame, like kill a boss monster, take out a large troop of enemies, or figure out a puzzle, you get a similar reward. They both try to create an intense experience while tying into that the emotional aspect of winning or losing. Even though videogames are almost unlimited in the directions you can take them, you still need to set up rules and restrictions to make the game a fun, focused experience.

GD: How did the Havok 2 middleware help?

BE: It provided a great foundation for us to make the world feel real and solid. It allowed us to quickly implement code for a water volume effect that includes force currents, implement that into the game, then use telekinesis to throw a dead enemy ragdoll into the water, have them float around and actually follow those currents as you would expect. On top of that, you can even shoot the body and see the impact of the bullet impulses on it (they get pushed around), all while the body continues to interact nicely with the water. Or, you can land a crate on them, and they'll sink. The combinations are endless, and there's no way we would've been able to get all that going in the timeframe we had without the solid foundation in place that using Havok gave us.

GD: How can future games take advantage of realistic physics?

BE: Over the past few years graphics have been a big focus in 3D games. Now, one of the next big areas to tackle in order to bring more realism into games and make them feel more immersive is physics. Imagine a world where given enough power you could destroy everything in a room as if it were a real structure in real life. Knock down enough walls or support beams and the roof or even the whole building collapses. Start a fire in one corner of a room and eventually it burns out of control so that the whole room goes up in flames causing the collapse of the building section by section so that you have to try to escape. At this point we don't quite have the processor power to make something like that happen as a real physics simulation. However, with the recently reported cell hardware design of the next generation platforms we'll start seeing this kind of stuff become a reality and great physics will become an entry level bar that games will need to succeed. Graphics will continue to improve, but physics simulations will become increasingly important, not just as a wow factor but as a part of the game design and experience themselves.

GD: What games are you playing now?

BE: Not much at the moment, as we are focused very hard on getting the game done. I did enjoy HALO, DEVIL MAY CRY, and GTA3.

Understanding Slerp, Then Not Using It

Recall that rotations can be represented by unit vectors in four-dimensional space, which have an algebraic structure. These 4-vectors are known as the quaternions, and they're especially useful for rotation interpolation problems, of the sort required by animation and inverse kinematics systems.

An infinite number of paths transition from rotation A to rotation B, but in general there's only one "straightest" way to get there, known as the torque-minimal path. Since this path travels along the surface of the 4-sphere, it is inherently curved. The function slerp walks along this path at a constant speed; slerp was popularized in the computer graphics industry by Shoemake (see For More Information on page 17).

Though quaternions are important, few game programmers understand how they work; the slerp function, in particular, is a mysterious black box. (Shoemake does not present a derivation in his paper; he just gives the formula). Certainly you can derive slerp by considering the geometry of a sphere in 4D space, but such spheres are difficult to visualize. So I'd like to present a derivation of slerp that works differently.

Slerp Solutions Are Coplanar

The inputs to slerp are two quaternions, q_0 and q_1, as well as a scalar parameter t that tells us how far to interpolate between these. ($t = 0$ gives us q_0, $t = 1$ gives us q_1, and intermediate values of t give us quaternions on the path between q_0 and q_1).

The first useful thing to realize is that all results from the slerp must lie in the plane P, defined by q_0, q_1, and the origin. I won't present formal proof here, because I want to get to the main proof. But if you'd like a formal explanation, see Genevieve Walsh's paper, section 1.1, titled "Geodesics in the three-sphere."

Because the hypersphere is symmetric in all directions about the origin, we can reflect it across any plane that passes through the origin, producing an equivalent hypersphere. We reflect the sphere across P. Because P contains q_0 and q_1, q_0 and q_1 stay in the same places. Since the rest of the sphere is mapped to itself, the overall geometry is the same. So the shortest path does not change. But since every point not contained within P is moved by the reflection, we know the path must be contained in P. (I have assumed here there is only one shortest path, which is implicit in the idea of constructing a well-formed slerp function; you can read about geodesics if you want to dig deeper into this subject.)

If this handwaving argument is confusing, you can just take it for granted that slerp always generates great circles of the hypersphere, and great circles are always coplanar (since they are circles).

How This Coplanarity Helps

Once we believe the results of slerp are all coplanar, the situation is still somewhat confusing, since the plane P is oriented arbitrarily in 4D space. But we can employ a simplifying strategy: suppose we build a transform that moves P from its arbitrary orientation onto the familiar XY plane. We can solve the interpolation problem in 2D space, then transform the result back to the arbitrary orientation in 4D.

Such a transformation could be represented by a 4×4 matrix. However, it would take extra CPU to compute and apply the transformation. Luckily for us, this transformation is unnecessary. If we feel like we need it, that's because we are overly attached to our coordinate systems. When we adopt a Zen-like detachment from the idea of coordinates, we find that the problem is simple.

Linear objects, like vectors and planes, exist independently from the methods we use to represent them (such as coordinate systems). These objects always obey the properties of linear algebra. Thus we can derive formulas directly from these linear properties, without ever mentioning coordinates—in the literature this is called a coordinate-free derivation. Formulas with coordinate-free derivations are very powerful; they must be true in all linear spaces, regardless of petty details like the total number of dimensions in the space (except perhaps for infinite-dimensional spaces, which are tricky and we should take great care when venturing into them, *hic sunt dracones* and all that.)

If you're not used to thinking in a coordinate-free way, Sheldon Axler's book *Linear Algebra Done Right* is a good start (see For More Information on page 17). For now I will show a coordinate-free derivation of slerp using 2D illustrations (which are all you need).

JONATHAN BLOW | *If you're looking for Jonathan Blow, you'd better check under the sea, because that's where you'll find him, underneath the ... or you could just send email to jblow@gdmag.com.*

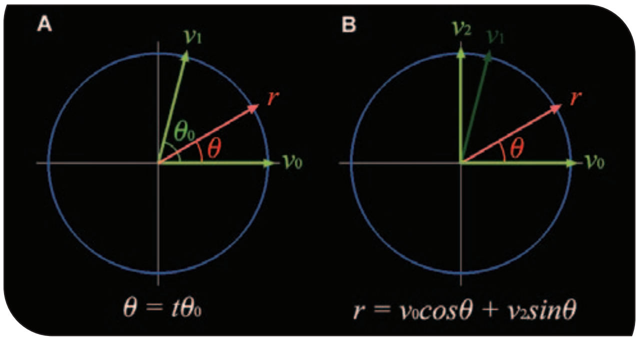


FIGURE 1: (A) We want to find the vector r that is at angle θ from v_0. (B) Using v_1, we build the ortho-normal basis { v_0, v_2}, which allows us to easily compute r.

TABLE 1: Rotation interpolation methods and the properties they satisfy.

COMMUTATIVE	CONSTANT	VELOCITY	TORQUE-MINIMAL
quaternion slerp	No	Yes	Yes
quaternion nlerp	Yes	No	Yes
log-quaternion lerp	Yes	Yes	No

Coordinate-Free Derivation of Slerp

Our inputs are two unit vectors, v_0 and v_1, and a scalar t. We are solving for a vector r, whose angle with v_0 is $\theta = t\theta_0$, where θ_0 is the angle between v_0 and v_1.

Figure 1A illustrates the problem. I have drawn v_0 in the direction we usually use for the X axis when drawing the XY plane. This by itself suggests a solution. If we had some vector v_2 that was orthogonal to v_0, as the Y axis is to the X, then our solution would be r = $v_0\cos\theta + v_2\sin\theta$.

Assume our input v_1 is linearly independent from v_0. (If it isn't, then the entire slerp problem is ill-formed; robust implementations of slerp contain a preamble to handle this case.) Since v_0 and v_1 are independent, we can ortho-normalize v_1 against v_0 to yield v_2, as in Figure 1B. That's it—we're basically done! The rest is implementation details, like finding θ_0 from v_0 and v_1. Listing 1 contains pseudo-code for the whole function. That's slerp—it's not some scary 4D thing. Listing 2 contains actual C++ source code. Though this code is written specifically for some type called a Quaternion, it is valid for vectors represented in an arbitrary number of dimensions; so if you have some dynamic n-dimensional vector class, you can just plug in the same source code. (Perhaps you want to interpolate surface normals on the unit sphere in R^3.)

Suppose we're not trying to be coordinate-free, but just want to solve slerp in the XY plane. Then finding v_2 is even easier; we can just say $v_2 = v_{0\perp}$, that is, $v_{2x} = -v_{0y}$, $v_{2y} = v_{0x}$. But if we then try and raise the problem to higher dimensions, we don't know what to do with this operator θ. It assumes we are living in XY coordinates, which is a relatively weak stance for an operator to take.

There's a cosmetic difference between this slerp and the Shoemake code. Where I use a Normalize function, Shoemake divides by $\sin\theta_0$. Indeed this has the same effect as Normalization; some trig will tell you the length of v_2 prior to normalization is $\sin\theta_0$, so the divide turns it back into a unit vector. I like to use the explicit Normalize, though, because it emphasizes the vector nature of the computation. As to which method is faster, that's unclear, as it depends heavily on the target hardware. But you shouldn't care, because if you're calling slerp in the first place, you are already in for a great deal of slowness; small deviations in that slowness are not going to matter much.

In fact, game programmers shouldn't be using slerp as often as they do. I think it's an important function to understand—hence this article—but a deep understanding of the function implies you know when not to use it, which is most of the time.

Alternatives to Slerp

As discussed in the paper by Muratori and Bloom (see For More Information on page 17), there are three basic properties we often want when interpolating rotations: commutativity, constant velocity, and minimal torque. Unfortunately, it seems impossible to get all three at once. There are three major methods of quaternion interpolation, and each of those methods gives you two of the three desirable properties. The choices are: quaternion slerp (popularized by Ken Shoemake), normalized quaternion lerp (which I will call nlerp from now on), and log-quaternion lerp (also known as exponential map interpolation, see the paper by Grassia). Quaternion lerp was discussed in the Shoemake paper but not considered important there. It was popularized by Casey Muratori, and some of us consider it very important in games now. See Table 1 for a handy summary of the properties of each interpolation method.

Currently, slerp is considered the authoritative method for rotation interpolation. This is because most programmers don't understand slerp, much less the alternatives; they just hear from other people that slerp is the right thing to do, then they paste the Shoemake routines into their source code. Shoemake presented the concept well and his paper was very helpful and relevant. But as Table 1 clearly suggests, we have several available solutions, so we should choose the one that fits our problem best.

Right now there are a few major tasks in games for which we use rotation interpolation; they involve mainly animation interpolation and inverse kinematics. (Camera control, for example, can be viewed as a sub-problem of these.)

When building animation systems, programmers often use slerp to generate rotations in between keyframes, then attempt to optimize the slerp function so that the animation system runs faster. For examples of slerp optimizations, see the Inner Product article "Hacking Quaternions" (March 2002) or Thomas Busser's article "PolySlerp" (February 2004). Generally these optimized slerps are slower and harder to understand than the normalized linear interpolation. We should only be using them if we really need our rotations to interpolate at constant velocity.

But as Casey clearly pointed out to me, continuity and runtime efficiency are the most important issues for an in-game animation engine, and nlerp delivers these the best. It's easy to get caught up

LISTING 1: DERIVATION OF SLERP

```
; Inputs are: unit vectors $v_0$ and $v_1$, scalar t
; $v_0$ and $v_1$ are linearly independent
Let $\theta_0$ = acos($v_0 \cdot v_1$)
Let $\theta$ = t$\theta_0$
Let $v_2$ = Normalize($v_1 - (v_0 \cdot v_1)v_0$)
return $v_0\cos\theta + v_2\sin\theta$
```

idea that slerp is the right answer and to worry about nlerp's nonconstant velocity, but in reality the issue is unimportant. Slerp is not the right answer unless the animator actually used quaternion keys with order-1 interpolation when authoring the animation, which is usually not the case. Unless we want to try to duplicate all of the quirks of an animation package's possible rotation representations (which we don't), we usually export animations to our game by sampling them at regular intervals from the art package (perhaps sampling at 30Hz, with the samples chosen in some convenient representation, like quaternions); then we perform some compression on those samples (like spline fitting) and save the result to disk.

Animators don't even know what their animations look like for time values between these 30Hz samples (or whatever frame rate they authored the animation at). Since the animator isn't intentionally authoring poses at those time values, it's silly to try and duplicate those poses.

Because of this lack of intentionality, at runtime we're concocting whatever rotation path we want to fill between samples. In this context of just making stuff up, there's no reason to spend extra CPU on slerp since we don't benefit from this expenditure. (To see why it's pointless, imagine the animator used Euler keys, in which case the "right answer" is some path that's non-torque-minimal, with non-constant velocity, certainly nothing like what slerp would give you.)

In fact, in a highly compressing animation export system, the nature of the visible animation is controlled at a high level by the spline fitter. The spline fitter is inherently going to adjust for the properties of the low-level interpolator by introducing and adjusting the knots of the spline until acceptable perceptual error

FOR MORE INFORMATION

Sheldon Axler, *Linear Algebra Done Right*, 2nd ed., Springer, 1997.

Si Brown, "An Introduction To Representing Rotations In Quaternion Arithmetic." www.sjbrown.co.uk/quaternions.html

David Eberly, "Quaternion Algebra and Calculus." www.magic-software.com/Documentation/Quaternions.pdf

F. Sebastian Grassia, "Practical Parameterization of Rotations Using the Exponential Map." *Journal of Graphics Tools*, Vol. 3.3, 1998. http://graphics.snu.ac.kr/OpenGL2003/10(1112)/expmap.pdf

Casey Muratori and Charles Bloom, "A Paper About Rotation Interpolation That We Will Never Finish Because We Are Lazy," referenced in gdalgorithms mailing list. www.gdalgorithms.org/archives/2003-05/459f4ae23eb2f34f.html

Ken Shoemake, "Animating Rotation with Quaternion Curves," *Computer Graphics*, Vol. 19, No. 3, July 1985.

Genevieve Walsh, "Great Circle Links in the Three-Sphere," Ph.D. dissertation from U.C. Davis. www.ma.utexas.edu/users/gwalsh/dissertationfinal.pdf

LISTING 2: C++ CODE FOR SLERP

```
// slerp(): v0 and v1 should be unit length or else
// something broken will happen.
Quaternion slerp(Quaternion const &v0, Quaternion const &v1,
double t) {
    // Compute the cosine of the angle between the vectors.
    double dot = dot_product(v0, v1);

    const double DOT_THRESHOLD = 0.9995;
    if (dot > DOT_THRESHOLD) {
        // If the inputs are too close for comfort, linearly
        // interpolate and normalize the result.
        Quaternion result = v0 + t*(v1 - v0);
        result.normalize();
        return result;
    }

    // Robustness: Stay within domain of acos()
    Clamp(dot, -1, 1);
    // theta_0 = angle between input vectors
    double theta_0 = acos(dot);
    // theta = angle between v0 and result
    double theta = theta_0*t;

    Quaternion v2 = v1 - v0*dot;
    // { v0, v2 } is now an orthonormal basis
    v2.normalize();
    return v0*cos(theta) + v2*sin(theta);
}
```

reached. So the exact properties of the runtime rotation interpolator don't really matter, so long as the path is not discontinuous or harsh. Thus nlerp is really the best choice for these cases.

For inverse kinematics problems, such as example-based IK, we tend to write iterative routines that solve for the goal rotation. Such a routine doesn't even try to hit the right answer on the first iteration (it can't do that), so the constant-velocity property of slerp is not useful. So long as the interpolation is monotonic, like nlerp, then the solver will find the goal without undue trouble. For example-based IK solvers, and for animation systems that blend more than two animations, commutativity is a highly desirable property, since it ensures that the results of the blend do not depend on the order in which the poses are mixed. Slerp does not provide commutativity, but nlerp does. Commutative blends are much easier to understand and work with. So nlerp is the best choice for these tasks too.

Don't Slerp

Game systems are big and complicated, so there may be occasions where you really do want to use slerp. But right now, I really can't think of any; so I encourage you to reconsider the use of slerp in your game if it's in any danger of causing speed problems. Fast slerp approximations add complexity to your engine and increase the difficulty of understanding the whole system, so they should be avoided when nlerp suffices.

Twist and Shout: Fixing Twisted Deformation

In "A Joint Effort" (Artist's View, February 2004), we introduced the fix-up bones as a means of combating the regrettable state of conventional real-time skin deformation. Fix-ups work by reproducing a fraction of the rotations in an animating skeleton. Assigning vertices to the fix-up bones, rather than directly to the animation skeleton, prevents the worst forms of vertex collapse around problematic joints like shoulders. This month we're going to look at some advanced fix-up strategies for a different class of deformation problems.

Sick and Twisted

The shoulder is clearly the worst villain of skin deformations, but there are many runners-up with different kinds of deformation failures. The most notable contenders are twisting joints. Necks, forearms, and biceps can all shrink or even turn inside out when twisting around their long axes. The problem stems from the same source we discussed in February. As with ball-joints, the skinning algorithm interpolates vertices linearly between their original and newly rotated positions. Naturally the rotation of the vertices takes place along an arc. The interpolation, being linear, always cuts a chord on that arc, slicing deeper into the volume of the model as the twist increases. At 180 degrees the chord passes right through the origin of the arc—in other words, the bone—collapsing all of the vertices down to a point (Figure 1).

The cure for this nasty hourglass effect is to adapt the strategy we used for the ball-joint fix-ups described in the February column. Twist fix-ups rotate around the long axis of a twisting bone by some fraction of the real rotation. They carry their assigned vertices to positions that reflect the correct rotational interpolation. This preserves the volume of the geometry and prevents the hourglass effect.

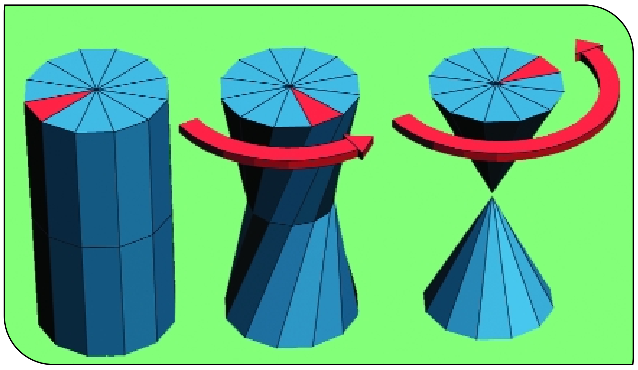


FIGURE 1. The collapsing effects of twisting deformations.

Doin' the Twist

Unlike ball-joint fix-ups, twist fix-ups only work in a single axis. This means they have to be controlled by expressions rather than constraints. So before we can build twist fix-ups, we need some way of representing the twist as a number that can be fed into an expression. Unfortunately, twisting (like obscenity) is easy to recognize but hard to define. Consider the arm of a typical animation skeleton. Intuitively we know that the twist of the arm is the rotation around the axis between the shoulder and the elbow. Naturally you'd assume that the Euler-angle rotation value for the axis that points along the biceps would be the twist value. The reality, alas, is a bit more complex. Figure 2 shows three sets of F-curves, which produce identical animations. The only difference among them is the order in which the Euler rotations are executed. As you can see, different rotation orders produce remarkably different results.

How can you get a good twist value then? The source of the problem is that each rotation influences the meaning of the numbers that come after it. Since each rotation changes the meaning of the next, only the first term in an Euler sequence has a stable meaning. Therefore, it's possible to determine the twist of a bone with Euler angles only if the twist axis is the first axis in the rotation order. Before trying to build twist fix-ups, double check the rotation order on your bones to be sure the axis which points down the bone is the first Euler term (Figure 3). In both Max and Maya the skeleton tools usually create bones this way by default, with the *x* as the twist axis and the rotation order set to the conventional *x-y-z*. But be sure to check! Your defaults may have changed, and building skeletons out of disjointed parts can cause unpleasant surprises.

The bad news about deriving twist values from Euler angles is Max's animation-controller architecture makes it impossible to get consistent Euler values from bones driven by IK, constraints, or TCB rotations. For IK-driven bones, the twist angle value on

STEVE THEODORE | *Steve started animating on a text-only mainframe renderer and then moved on to work on games such as* Half-Life *and* Counter-Strike. *He can be reached at stheodore@gdmag.com.*

the IK handle is often an adequate substitute and can be fed to twist expressions. Generally, though, the only alternative for Max users is to create a custom twist attribute and animate it by hand. Ferreting information out of Max controllers for expression and script writing will be the subject of a future column.

Everybody do the Twist

Once you've succeeded in abstracting a twist value from your bone, adding twist fix-ups is pretty easy. Let's look at a common example: a one-link neck (Figure 4). The fix-ups themselves are merely extra bones laid out along the length of the neck. Fix-ups always form a parallel branch of the animation hierarchy rather than an in-line part. Here the fix-up bones are all children of the neck bone, thus, siblings of the head and also of one another. As siblings of one another, the numbers we feed into their twist expressions will have consistent meanings. Before the expressions are added, the fix-ups' rotations should be zeroed out so they start off aligned exactly with the neck. Each fix-up should be matched to a ring of vertices in the cross section around the neck. In this case three fix-ups seems sufficient to keep the neck from collapsing.

In this example we have three fix-ups, located at the base of the neck, one-third of the way down, and two-thirds of the way down, respectively. Each fix-up gets an expression of its twist axis, assigning a rotation proportional to the fix-up's position along the length of the bone. We don't want the first fix-up bone (the one at the base, located at zero-third of the way down) to twist at all. The fix-up at one third of the way will rotate 33 percent of the total twist, while the one at two-thirds of the way reaches 66 percent. The remaining 33 percent of the total twist rotation is handled by assigning vertices directly to the neck bone itself, which of course rotates by three-thirds of the way.

The subtlety here is that the fix-ups are actually counter-rotating, reversing the twist of their parent, the neck bone. Therefore, the first fix-up rotates backwards by the whole twist value, the second rotates negative two-thirds of the twist, and so on. The mathematically-minded may express this as follows:

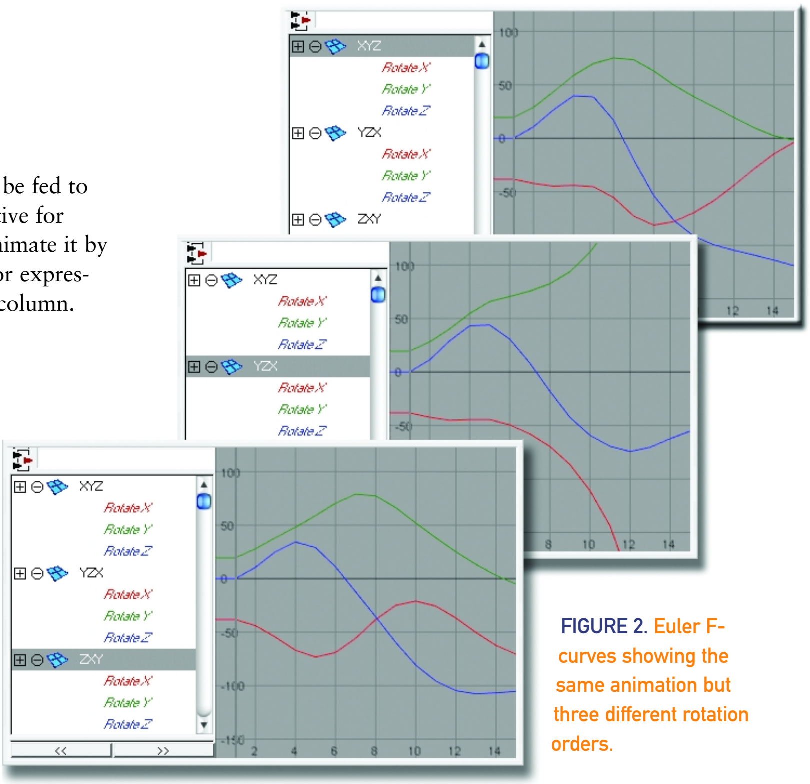
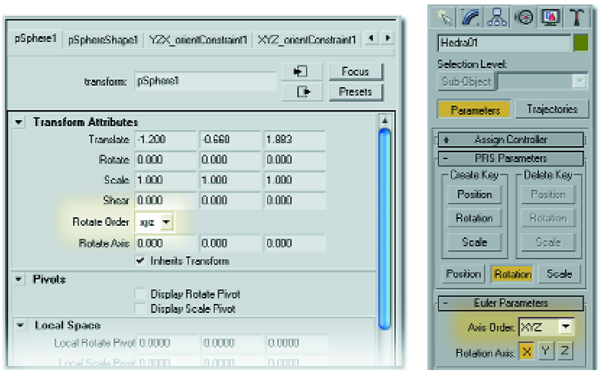


FIGURE 2. Euler F-curves showing the same animation but three different rotation orders.

Fix-up rotation = (1 – the fix-up's position a fraction of the bone's length) * (–1 * the bone's twist).

As you can see, the setup is very simple and can be driven by kindergarten-level expressions. If all you're after is avoiding twist-induced collapses, that's all you need. If you're feeling adventurous, you can tweak the distribution of the twist by changing the rate at which the fix-ups rotate. The previous example is a simple linear distribution: each fix-up's rotations are mapped directly to its positions along the bone. You could compare this to the behavior of an F-curve with linear tangents. Changing that relationship will distribute the twist differently. For example, if the twist expressions were set up as (1 – the fix-up's position over bone length, squared) (–1) (bone twist), you'd get a slow start and a sharp end to the distribution of the twist—in other words an ease-out F-curve. Don't be afraid to experiment if you have a particularly difficult piece of geometry or motion. Don't obsess over the math. There's nothing wrong with skipping the formulas and just fiddling with different constraints to get good-looking results.



transform: pSphere1

Translate	-1.200	-0.660	1.883
Rotate	0.000	0.000	0.000
Scale	1.000	1.000	1.000
Shear	0.000	0.000	0.000
Rotate Order	xyz		
Rotate Axis	0.000	0.000	0.000
Local Rotate Pivot	0.0000	0.0000	0.0000
Local Scale Pivot	0.0000	0.0000	0.0000

✓ Inherits Transform
Display Rotate Pivot
Display Scale Pivot

Hedra01
Selection Level: Sub-Object
Axis Order: XYZ
Rotation Axis: [✓ X] Y Z

FIGURE 3. Where to check rotation orders in Max and Maya.

Let's Twist Again

So far we've tried using counter-rotating fix-ups to remove parts of the rotation from the base of the neck. Often, though, you may want to put twist fix-ups onto a bone that is not itself twisting. Our hypothetical neck bone might be set up with its twist rotation locked, and the twist value could be derived from the rotation of the head instead. The fix-ups will work equally well at preventing collapses. Using a downstream bone to drive the fix-ups involves only two changes to the arrangement we've already described.

The first difference is that the distribution of the fix-up bones is reversed. In our example, we positioned fix-ups starting at the bottom of the neck, since we wanted to remove the twist at the base. If the neck is not twisting, though, we can eliminate the fix-up at the base and assign the vertices there directly to the (untwisted) neck bone. At the same time we'll need to add a fix-up at the upper end of the neck bone that will handle the three-thirds rotation previously left to the neck bone.

The second difference is in the expressions. When the neck was twisting, we needed the fix-ups to counter-rotate against that twist. Now that the neck is not twisting, the fix-ups rotate positively. The fix-up one-third of the way along rotates one-third of the twist value, the fix-up two-thirds of the way along rotates two-thirds, and so forth. The new expressions can be summed up as follows: fix-up rotation = twist times the fix-up's position as a fraction of bone length.

You may ask, why drive the twist with a different bone? This technique is particularly useful for fixing troublesome forearm deformations. Although we conventionally rig characters with ball-joints in their wrists, in reality the wrists have only two degrees of freedom. The twisting motion actually comes above the wrist, from the torsion of the ulna and radius bones in our forearms. Long sleeve can disguise the problem, but bare-armed characters are often afflicted with pinched, robotic wrists or rubbery Gumby-style forearms. Splitting out the forearm twist into a separate bone is cumbersome for FK animation and problematic for IK as well. With twist fix-ups driven by the wrist, though, it's easy to get reasonably good forearms without changing the animation rig at all (Figure 5).

Twist Off

The ultimate challenge for twist fix-ups is our old nemesis: the shoulder. As we have lamented in the past, shoulders are very tough to skin effectively. With a twist range of nearly 180 degrees, shoulders also cause nasty shearing in the upper arm. It doesn't help that the game industry's steroidal mindset creates a lot of characters with gigantic biceps that magnify

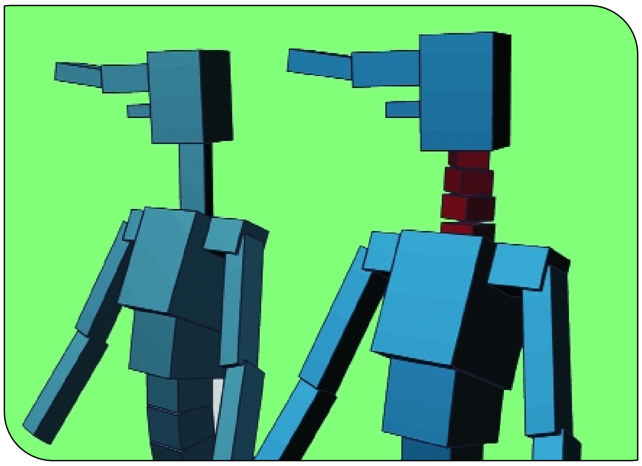


FIGURE 4. Compare the difference between a single-joint neck and the a neck with twist fix-ups in action.

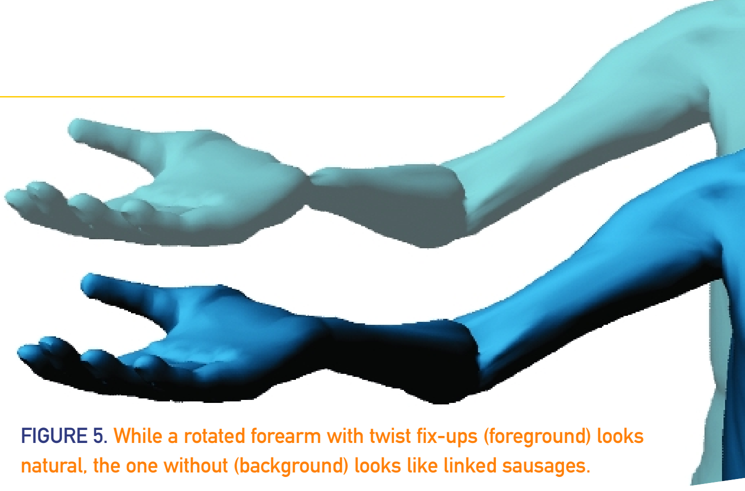


FIGURE 5. While a rotated forearm with twist fix-ups (foreground) looks natural, the one without (background) looks like linked sausages.

every twist problem. When we introduced ball-joint fix-ups in February, we noted they can preserve the volume of a collapsed shoulder but don't protect against biceps shearing. Now we can rescue the biceps as well by combining the best aspects of twist and ball-joint fix-ups.

Relaxed Shoulder

A hybrid ball-and-twist approach starts very much like our first example, by adding twist fix-ups to the biceps bone. These fix-ups will rotate backwards against the twist of the biceps bone, as in the first example. The fix-up located on the shoulder, though, works rather differently. Instead of being parented to the bone, it's parented to a ball-joint fix-up, also located at the shoulder. The ball-joint fix-up will produce a weighted average of the shoulder's and clavicle's rotations. The twist fix-up then needs to counter-rotate against the twist component of the shoulder's rotation to remove the twist from the base, just as in the neck example. To do this, the twist fix-up's expression will equal the biceps' twist times –1 times the weight coefficent of the ball fix-up. When assigning vertices, simply ignore the ball-joint fix-up and use only the twist fix-ups. The result of this should be to spread the twist of the arm along the biceps without losing the rotational compensation of the shoulder fix-up (Figure 6). This goes a long way towards preventing the worst shoulder deformation atrocities. The fix-up bone technique we've described here is still only a crude approximation of what really goes on under the skin of a real person. There's no denying it's a hack; nevertheless, it's a pretty useful one and, despite the large chunk of text it takes to describe it, not really very difficult to implement. The biggest handicap is the difficulty of extracting twist data from Max controllers, so if anyone out there has developed good ways of getting those Max numbers, feel free to send them to me. I'll share them when we return to the topic of using Max bones with expressions a few months later.

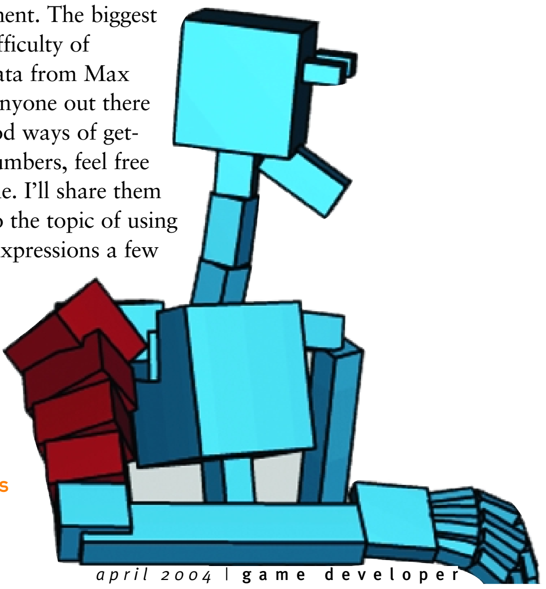


FIGURE 6. A hybrid ball-joint and twist fix-up for easing shoulder and biceps collapses.

Stand Out and Be Heard

Game music has changed immensely in the past few years—perhaps more so than any other aspect of game development. One of the most alarming changes is the tremendous number of people getting into the business. What was once a fairly niche market is now mainstream: game music has hit even *Keyboard* magazine (www.keyboardmag.com). So, how do you stand out and get noticed?

Ask game composers and sound designers what their job is and they'll most likely say, "to create the sound/music for games." Wrong.

1. Remember your job. Ask game composers/sound designers what their job is and they'll most likely say, "to create the sound/music for games." Wrong. You're hired to make the game better. Towards that goal, you create music and sound effects, certainly. But remember: first and foremost, your job is to make the game more fun, not to show off your composition or sound design prowess. That may mean toning down the music or cutting your favorite orchestration because it doesn't fit the game.

2. Get technical. Know the bits and bytes of game audio. See the world from the programmer's eyes. Do you know every way the system can make a sound? What about the impact that a particular method has on the rest of the system? By knowing the systems inside out from an audio perspective, you'll be the leader in creating the audio technical specifications and delivering a top-notch soundtrack.

3. Know the publishing scene. Educate the producer about how to publish game soundtracks and be able to rattle off the financial benefits they can reap from your work. If your soundtrack is nominated for a Grammy, you can be sure you'll be on their short list for their next project.

4. Know your tools. Know what tools are available from either the console manufacturer or third party middleware companies. Become an expert in all of them. This part is important: practice using them before your first job on the system so you hit the ground running.

5. Have a full arsenal at your disposal. Be able to create music and sound effects in a wide variety of formats: Studio-generated effects, MIDI-generated effects, parameterized sound effects, and more. Know every way the game system can create sound and be able to exploit the options as appropriate.

6. Make the programmers' job easier. Make sure you are working with clear specifications. Have the delivery format and methodology spelled out. Deliver crystal clear instructions to the programmers and schedule time with them to assist in implementation. By saving the programmers time and headaches, you'll impress the producer, which leads us to the next point.

7. Make the producer's job less stressful. Keep the producer worry-free when it comes to sound. Keep the big picture in mind. (See #1). Don't be afraid to challenge him or her on creative issues, but know when to push and when not to. Remember, audio is but only part of what the producer has to think about. Deliver the work on time, in the right format.

8. Beware of the dark side. When creating audio for multiple platforms, it's often easiest to work to the lowest common denominator, the set of features common across all of them. While this is the quickest, if you do this, you may end up compromising quality, and therefore reputation. A few small platform-specific tweaks can make a big difference.

9. Look for non-traditional niches. Ring tones, mobile games, and web sites all represent technologies looking for innovative, interactive audio. Though not as sexy as composing for a live orchestra for a next-generation console, these gigs can add significant income to your bottom line, not to mention the networking possibilities.

10. Push the boundaries. Be a leader in the industry. See what interactive sound and music tools there are and use them. See if some physical modeling might be appropriate. Show the team you know game audio, you get game audio, and they'll look to you for leadership in this (and future) projects.

Game audio is still a new and growing field. By shining in these areas, you'll set yourself apart from the legions of composers and sound designers who think all there is to game audio is creating some WAV files and putting them on an FTP site. Let's make sure it's your phone that rings the next time someone needs sound for his or her latest game.

BRIAN SCHMIDT | *Brian currently serves as program manager for Xbox audio and media at Microsoft. A member of the Game Audio Network Guild's board of directors, he has been in the multimedia audio industry since 1987. He has composed music for more than 120 interactive games, including* JOHN MADDEN FOOTBALL, JURASSIC PARK, *and* STAR WARS TRILOGY.

Have Cell Phone, Will Play

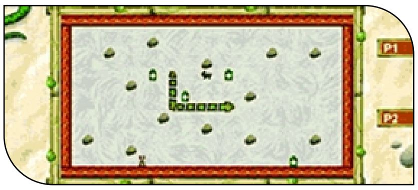
Games for mobile phones have been a hot topic for several years now, with lots of interesting development around the world. So this month, we'll examine some mobile game design rules.

Greg Costikyan is a designer with considerable experience. With a successful career in the paper game community, he went on to co-found West End Games, spend many years at Crossover Technologies in early MMORPG development, and, most recently, co-found Unplugged Games to explore wireless game design. At a GDC tutorial in 2002 Greg offered the following rule:

Design to the medium's strengths instead of struggling with its limitations. It's similar to my own "Begin to design by identifying your constraints," which I shared in July 2003 ("2 for the Design Process," Better by Design), but I like Greg's variation more. That's a great rule for any game, but is particularly relevant when we consider a new gaming medium like mobile phones. I'm accustomed to using the term "platform" to characterize the device running a game, but as Greg's choice of "medium" is more accurate and elegant—not every game needs a computer to play.

But what does this rule mean? Specifically, Greg suggests we consider the advantages of a cell phone: it's portable, people carry it everywhere, and it's connected to other people. Based on those features, focusing on simple games that can be played quickly is a good idea.

I found a Postmortem on Gamasutra, "Ngame's Chop Suey Kung Fu" by Matt Kelland (www.gamasutra.com/resource_guide/20010917/kelland_01.htm), to confirm this view. He notes that Ngame had other more ambitious games, but by creating a simple game that takes advantage of dice-based games (yet another game medium!), the company had one of its biggest success-



Snake Ex (for Nokia Communicator 9200 series) can swallow your idle moments.

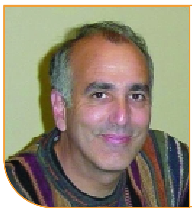
es. I also found echoes of Greg's rule in another Gamasutra article, "Designing Mobile Games for WAP" by Lasse Seppänen (www.gamasutra.com/resource_guide/20010917/seppanen_01.htm). Lasse says it's critical to know the "limits and opportunities." Let's face it—if you have access to a better medium to play games, you won't use the cell phone. So you need to design games that can be enjoyed in short, idle moments. That's an insight reinforced by my experience. I don't usually consider playing games on my phone when I have an alternative; my portable Game Boy, for instance, makes a more preferable system. But there are times when I have a few minutes to spare and nothing but my cell phone to keep me occupied. Greg also confirms that, in his experience, compact games for casual gamers have been more successful than full-featured games for hardcore gamers.

Design the game to fit the revenue stream. That's a rule suggested by several designers. The revenue model is similar to the old boxed game model; we've grown so accustomed to retail distribution we often forget there are alternative methods. Certainly makers of Internet-distributed games and MMORPGs have had to shift their thinking to maximize their revenues.

For mobile games, the game should be quite different depending on how it intends to earn money. A game designed for a hardware company to be built into the phone won't earn money by using the networking capabilities of the phone. Besides, it'll take up valuable space on the phone's ROM. So we end up with simple single-player games like Snake. A game sold through a specific mobile carrier could maximize phone calls or SMS data by hooking the player with repeat play, because the more people use their phones, the more money the carrier makes. A game created by an independent developer should be designed to work with as many phones and carriers as possible, so it won't alienate any potential market.

I'm personally intrigued by some of the emerging technologies getting incorporated into mobile phones and PDAs. One such technology is the global positioning system that allows the game to know where the player is physically located, possibly playing with others in the same physical vicinity, at a party or a rock concert for example. Such games can, for instance, require the players to change their locations during the game. Bluetooth and other local networking features would facilitate some interesting trading games modeled after Magic: The Gathering or Pokémon. For instance, trades can be made automatically when other players are nearby.

The possibilities are intriguing. As in combat, the one who can mobilize quickly and effectively is most likely to emerge as the victor.



NOAH FALSTEIN | *Noah is a 24-year veteran of the game industry. His web site, www.theinspiracy.com, has a description of The 400 Project, the basis for these columns. Also at that site is a list of the game design rules collected so far, and tips on how to use them. You can e-mail Noah at nfalstein@gdmag.com.*

Return

Illustration by Steve Munday

The Wireless Gold Rush

It has been five years since wireless games first started popping up on the radar of game developers, first as a curiosity, and then as a potentially real business. This time last year, the big conference room at the Game Developers Conference was jam packed with developers wanting to know the scoop and being amazed at the state of intriguing confusion they found. Here was a market with hundreds of millions of potential customers, a micro-billing model that actually seemed to be working, and some pioneering kindred developers who actually seemed to be wearing smiles during an otherwise painful time in the game industry. On the other hand, there were no good numbers to be found and few publishers. Instead there were carriers who wanted to look only at completed titles, cherry-picking what they liked and leaving the rest like so much scrap wood on the shop floor. Now that a few years have passed, has the smoke cleared? Do we know enough now to make more reasonable investment decisions? And have we waited so long that all the good claims are taken? Is the gold rush in full swing, or are there just a few played out streams left?

Where Things Are

A few years ago, Nokia befuddled the developer community by showing up with a booth at GDC to promote, of all things, cell phone games. They gave away developer kits and showed off black and white mini-screens with such monster titles as SNAKE. SNAKE as it turns out is hugely important to the Finns who run Nokia. It was stuck on to one of their phones in 1997 as a lark by someone in one of their engineering groups. They were shocked to see how popular it became, and it hooked our cold-weather friends on the concept of gaming. Very few in the gaming community realized how significant they were. Nokia is to mobile phones what Microsoft is to software. Their little SNAKE game ended up on more than 250,000,000 handsets before the end of the second year.

It is important to realize the psychology of this. Nokia, like Microsoft, is an engineers company. They discovered games as engineers do, with great excitement, and as it turns out, absolutely no historical context. These were not the people who grew up on the Game Boy, Lynx, Turbo Express, and Game Gear. They lived through neither the great successes nor debacles that have colored the current generation of game developers. They come with the wide-eyed innocence of the unbeaten, and the frightening strength of a giant. Their strength, however, was hidden from those with a U.S.-centric perspective. And so we laughed at this goofy company trying to get us to take cellular phones seriously while we were trying to see which platform was going to give us the most textures and polygons.

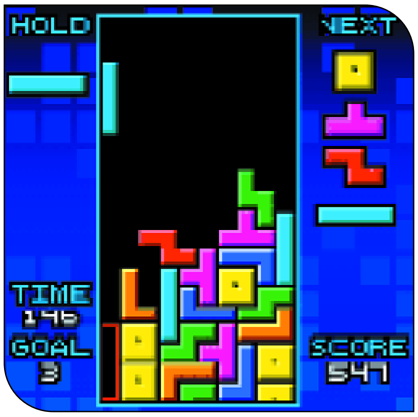
Part of the massive lack of interest had to do with the rise and fall of the WAP standard as a way to distribute and play games via cellular phones. A number of game developers, including later cellular game superpower, Jamdat, had massively underwhelming customer results with this first generation technology. (There are still some, by the way, who swear by the royalty checks they are collecting for WAP games even in current time.)

The next place mobile games started to pop up on the radar was in Japan. DoCoMo, the country's largest mobile service provider, had introduced a service called I-Mode in 1999 and they were successfully selling games as applications using micro-payments charged directly to a customer's cellular bill. This was both the long promised micro-payment scheme in action and a pay-for-download model that was actually working. There were rumors of massive successes, but that was there and this was here, and there were lots of rationales why it wouldn't work here. They followed this in 2001 with the launch of their I-Appli service, concentrating on monthly subscriptions of between 100 yen and 300 yen, rather then one-time download fees. Major Japanese videogame houses from Namco to Hudson started shipping cellular versions of their old games.

In 2002, Sprint and Verizon brought out their own versions of DoCoMo's model, and the mobile phone videogame market hit U.S. shores. AT&T quickly followed, and a cavalcade of greatest hits from the early 1980s moved from the Atari 2600 and equivalent on to cell phones. This shovelware approach didn't have as chilling an effect on the new market as one would have thought because of an interesting coincidence. The demographic of folks who could

BEN CALICA | *Ben is a frequent contributor to Gamasutra.com and* Game Developer's *sections on wireless gaming. In past lives he was director of production for Cyberflix Games, where he worked on LUNICUS, JUMP RAVEN, and SKULL CRACKER. He also played a key role in creating Apple's Game Sprockets technology.*

afford the $300 mark for the high-end phones that could play the first set of games tended to skew into the 30s to 50s in age. Exactly the group who had played the 1980s games in the first place. Those games were the first hits, plus golf and casino games, all of which were of interest to the boomer and boomer plus generation. By 2003 the business was in full swing, with Verizon announcing they'd had more than12 million downloads for their phones, the majority being games and ringtones. The game was clearly on.



Blue Lava's mobile version of Tetris, dowloadable on a Sprint PCS handset.

The Politics of Mobile Games

One of the greatest challenges in wireless game development has to be the clash of cultures between the game and cellular phone worlds. Beyond a well documented clash of even basic terminology, ("platform" means completely different things in the two lingos), there is huge difference in the basic nature of who rules the roost and how each group deals with the other. The basic model in the game world is that great games are the wood that fuels the fire. The console makers fight like crazy with each other to get the best games on their platforms, hoping for an exclusive if they can get it. Developers take one of two approaches. They either bring their pitch and credibility to the table to attempt to get a publishing deal, or they bring their reliability so they can be the house that develops a great license for the publisher. In either case, they trade the risk the publisher is going to take in funding their development for a smaller piece of the final pie. And that risk is quite real. In addition to the development costs, the publisher is going to take the horrific risk of producing "the right number" of units for the all-important Christmas holiday. If they guess wrong, they sit on a large number of units that come complete with royalty pre-paid to the console manufacturer. Publishers also take care of distribution and advertising and every other little thing to push the game on to success.

The cellular world is utterly different. The carriers (Sprint, Verizon, AT&T/Cingular, and so forth) are the kings of the hill. They own the customers down to their toes and have grown up in a world that exists to serve them. The handset manufacturers, who include the aforementioned, ultra-powerful Nokia, bow to them like handmaidens. It is the handset manufacturers who are in the death grip with each other to show that they have the coolest toys on the block, and they are the ones who have been propelling the game business forward. But looking at the way they have set up their developer relations programs between the game developers and the carriers gives a pretty good clue to the nature of the relationship they are used to. Nokia's program, for example, provided a blind browsing area for developers to put up their completed wares for the carriers to stroll past and cherry pick. Get that? They wanted to have the developers put up completed work, done on spec, in the hopes they might get a publisher to pick them up. On hearing this, most traditional game developers just laugh. One of the big changes in the last year is in this area. The game developers who were first to market, developed good relationships with the carriers, and by the virtue of those relationships, stepped into the role of publishers. They will now pay advances to get a game developed, much the same way their console brethren do. They have less risk,

The Mobile Game Publishing Landscape

CARRIERS	MAIN HANDSET MANUFACTURERS	TOP 20 MOBILE GAME PUBLISHERS	
Verizon Wireless	Nokia	Jamdat Mobile	Hudson Soft
AT&T/Cingular Wireless	Motorola	Sorrent	Gameloft
Sprint PCS	Sony/Ericsson	Mforma	In-Fusio
T-Mobile/VoiceStream	Samsung	THQ	Sega
	LG	Sony Pictures Mobile	Blue Lava Wireless
	Siemens	Sony Online	Airborne Entertainment
	NEC	Entertainment	Mobliss (now Index)
	Sanyo	Walt Disney Internet	SK USA
	Kyocera	Group	Cybird
	Audiovox	Bandai	G-Mode
	Nextel	Namco	iFone

because there are no manufacturing fees, but they will give up one of their few precious publishing slots, and that is worth a great amount. They also often bring great expertise and tools used to deploy a title to many different handsets, a contribution not to be underestimated.

Digital Shelf Space

Part of the reason the carriers so clearly hold the cards has to do with who stacks the deck. In this case, it's the tiny set of nested menus on the mobile phones that represents the equivalent of the digital endcaps. While other countries, such as Japan through DoCoMo, have actually published printed catalogs with thousands of applications, the U.S. carriers have much more fear of overwhelming their customers, and so are actually looking to reduce the number of items on the deck from as high as 300 this year down to 100. The call is for quality and hits only. The other related challenge is the amount of space a developer has to get the attention of their customers. These are unlike the console or PC game business where there is a combination of well read magazines to use as a start and shelves to put boxes full of teaser screen shots and copy. In the mobile world you have a single sentence of about 30 characters to grab attention. "Retail space rules console game sales. Similarly, deck space rules wireless game sales. Keep your titles as short and as informative as possible," advises Centerscore CEO Oliver Miao.

This is part of why licenses are so important. If you have only five words, having one of them be "Hulk" helps quite a bit. The other tough factor is the "what's new" menu. It is gold while a game is resting on it, and can represent the end of sales when removed. One positive side is that each deck is customized to the particular model of phone. Most games do not run on all phones, so a developer can get more attention on a less popular phone if they want to increase their overall sales. "Since last summer or so, I've noticed a big difference in publishers' expectations," says Marcus Matthews, principal of Blue Heat Games, an experienced wireless game developer. "There's a huge flight to quality and licenses. There's enough feedback out now that poor quality and unlicensed games aren't selling well. It's so critical now, in some cases, publishers are spending more money on the license than developing the game. They're also raising the budgets for their games, understanding that high quality games need more time and recourses."

So What's a Winner?

This is a very tough question to answer. In the console and PC world, there are a number of analysts who have set up watch all over the industry and can give pretty accurate figures about sales. In the mobile world, a side-effect of all the carrier politics is that there is a culture of secrecy that goes all the way down to the developers and no one wants to be the first to talk. This caused great anger among the developers at the GDC Wireless Forum last year, but things have loosened up a bit. The ranges are still pretty wide, but here are some ideas about what a good title can sell. On the low end, "50,000 units at $3," according to Matthew Bellows, publisher of the wireless gaming site Wireless Gaming Review and the *Mobile Entertainment Analyst* newsletter. (More on that $3 later when we talk about the fun and frolic of pricing.)

The frustration about the lack of public numbers is felt even within the industry, such as at Qualcomm, the company that provides BREW, one of the two main delivery platforms for the industry. Mike Yuen, their director of developer relations, laments sell-through data is extremely tough to come by since carriers guard this info like gold in Fort Knox and don't readily share it. "We actually see this data on an aggregate basis in the BREW system among our carriers, but we can't share since the data belongs to the carriers and the publishers/developers and they would have to approve of us releasing it," said Yuen. It would be better if the industry had an industry yardstick for everyone to measure against for commonality, such as NPD Techworld.

On the other hand, Mitch Lasky at Jamdat Mobile is very good and open to providing public stats. In September 2003, he said he had 4.4 million game downloads on BREW, 12 percent of his paid downloads were JAMDAT BOWLING (the major wireless hit), and 10 of his titles had made more than $250K each. Do the math and you can see the ROI multiples are there for wireless games today. Scale will come once more and more of the installed base of consumers upgrade to over-the-air downloadable game-capable color handsets. This often takes a few years and I truly believe that day will come, probably sooner than later.

However, downloads aren't necessarily an ideal way to gauge whether a game is a hit. Currently the majority of new phones that are going out have the capability to download and play games, however, they represent a small fraction of the mobile phone user base. The industry calculates that phones get turned over in a period of about 1.5 to 2 years. That means the potential base of customers for cell phone games increases every month by an amount that is both regular and significant. That is part of why folks at companies like Jamdat have those wry smiles every time you talk to them about numbers.

Sennari Mobile's JAMDAT BOWLING, a major wireless hit.

How Much Does It cost?

Costs have been creeping up in the last couple of years for mobile phone game development projects, but they are still small devices with budgets and schedules a fraction of that of bigger game projects. This is still a world where a few folks in a garage can make a real-deal game.

According to Kevin Gliner, CEO and founder of Knockabout Games, "The range is still fairly broad: $30,000 to $70,000, 3 to 4 months. There are so many variables though, not the least of which is how many platforms to support: J2ME, BREW, or both. Symbian and Smartphone projects cost more as well." He continues, "A wireless game project is much more engineering-centric. For us, art takes about one quarter the time of our programming effort. On a traditional game project, the ratio is closer to even. The real limitation is the download package size and available heap space on the phone: art assets take a lot of space, so there's only so much you can do even if you wanted to spend more time."

Gliner goes on to speculate that the amount of time spent on art relative to coding should grow as high-speed networks become more commonplace and handset memory increases. He says, "Most wireless developers don't use common game industry tools as part of their art pipeline (such as 3DS Max). Without this expertise, they may struggle to keep pace as the demand for quantity and quality increases." Gliner takes a divide and conquer approach to working with the platforms. He says, "Our projects typically have one full-time engineer for the lead SKU (almost always BREW) for the entire project, plus a full-time engineer for the secondary platform (J2ME). Porting to J2ME sometimes takes as long as the original BREW version if we're really pushing the bleeding edge in terms of handset capability. Everything else is treated like a service group: art, game design, QA, and project management. That usually translates to quarter-time for those resources on each project." Blue Heat's Marcus Matthews

Wireless Developers' Advice.

Make sure the project makes sense for your business. Some deals may pay the bills, but add no long-term value to your company. Do you want to build a reputation as a strong sports developer? Then working on card games isn't going to further that goal.

Build the best game you can, not the best game for the budget. Or don't work on a title that can't be anything more than mediocre because of budget. No one cares if you did amazing work for the amount of time you spent on it. If all you do is mediocre products, then mediocre products is all you'll be offered.

Design for the limitations of the medium. Don't make dumbed-down versions of games from other platforms. Think about the core experience that makes that kind of game fun, and use that to design the game from scratch for the target handset.

Be wary about designing a game for the low-end and porting up. If you fail to take advantage of the strengths of the better handsets, your product is going to look weak on those devices. Likewise, designing for the high end and porting down will cause headaches if you haven't designed a game that can also work on a low end device.

– Kevin Gliner, CEO and Founder, Knockabout Games

First, making good deals with publishers is the same as with traditional game development (or any other entertainment medium). It's a function of the reputation and quality of the developer. It's very, very tough for a first time developer with no prior track record to negotiate a good deal.

Make sure you have enough of an advance to do a great game. There are a lot of hidden issues since no two phones are alike. Watch out for things like compatibility between the emulator and the device, graphic performance, input speed, and sound support. Those little problems can eat away at your time and increase your cost significantly.

Right now, it's much easier to develop on BREW devices because the hardware is more standard (all its phones use the ARM processor). But I don't see this problem abating until three to four years out.

– Marcus Matthews, Principal, Blue Heat Games

Expectations often exceed reality. When your publisher has a successful track record of launched titles, they will already have understood both the life cycle of wireless games development and the technical possibilities of a handset.

Never develop a game without getting credit information for your studio in the game.

Unless you've developed over half a dozen titles, it's a good idea to multiply the expected work by two. With cell phones, it's never as easy as you'd like it to be.

Utilize the help of big company developer programs—Qualcomm and Nokia have been the best for us.

– Oliver Miao, CEO, Centerscore

Understand that the mobile game environment, is just that. It's always changing at a rapid pace. Make sure your development plan and bid accommodate needed changes. Give the publisher some extras—an extra feature, an extra handset port, help them with testing, and so forth.

– Mike Cartabiano, President of Sennari Mobile (Jamdat Bowling) concurs, "The time and budget has gone up in the past couple of years. When we first started in fall 2001, the industry was transitioning from the text-based WAP days, where games cost up to $10,000. So naturally, the publishers wanted J2ME and BREW games done for the same amount." While publishers tend to be vague about advances, Matthews says that "today, a simple, low budget game, like a card or puzzle game, for BREW or J2ME would be around $30,000, whereas a complex action game with a robust feature set are running upwards of $100,000. And that's not counting the N-Gage, which supports 3D and is more like a PlayStation or Game Boy Advance, and has budgets of over $300,000."

He compares mobile development to that of Nintendo's Game Boy Color. "The Game Boy Color was a very limited device and, thus, did not need large programming and art teams. The typical mobile project supports a couple of people, with the art taking up a small part of the man hours."

"An original game will take us between one month to three months to develop for one platform," says Centerscore's Oliver Miao. "The average time for development is typically two months. Our longest game to develop to date, a GARFIELD pinball game, took four months. However, we plan on development time increasing dramatically this year as handset capabilities and quality expectations in the marketplace continue to rise. I would not be surprised to see development times double by this time next year. Depending on how involved and complicated a game is, our development costs now range between $20,000 and $100,000."

How Much Do I Charge?

This time last year, pricing was a huge debate. No one knew, including the carriers, what kind of pricing model would work for these games. So in the great model of decision by committee, the carriers and the publishers decided to put both a monthly price of usually in the $2 to $3 range and an unlimited download for two to three times that amount, just to see what would work. A year later and that has become, not an experiment, but the answer. Game developers hope for the holy grail of "subscribe and forget," and more often than not, at least for a few months, they can get exactly that. According to Blue Heat's Matthews, "I think it's too early to tell. People are driving revenue with one-time downloads and with subscription sales. However, I haven't noticed any breakthrough models yet."

Matthews extends his Game Boy Color analogy into offering potential for increased revenue per unit. "Now that there are games that rival the Game Boy Color, there is no reason mobile games couldn't be priced $10 to $20 if users have paid $20 to $30 for a top Game Boy Color title," he says. "I think you're going to see a tiered pricing model, similar to the PC market, where you'll have the AAA mobile games priced at the high end ($15 and up) and then unlicensed budget games at the $2 to $4 range. The top mobile publishers seem to have shifted to a quality versus quantity strategy, which is what has happened on consoles and the PC as they matured."

Qualcomm's Mike Yuen adds, "The ability for a publisher/developer to price with flexibility in my mind is critical. For example, being able to offer the user multiple purchase options and not just one." From his vantage point, "the most successful pricing schemes thus far have been one-time unlimited use and monthly subscriptions. In the future you'll see more incremental value billing for content like buying new levels or ancillary stuff like ring tones or screen savers or wall paper. You'll also see gifting, bundling, and the ability to do monthly subscription with upsell within the subscription."

What Does the Future Hold?

So what does the next two years look like in the cell phone world for game developers? "That's an issue of great concern," according to Blue Heat's Matthews. "Right now, there are just too many handsets for North American publishers and developers to support profitably. To get a reasonable penetration of the capable handsets, developers should support more than 40 devices. And that number is increasing every month."

However, as phones with better graphics and more capabilities hit the market, development industry will feel an even greater strain. As Matthews points out, "Until the industry or carriers starts to coalesce around a hardware standard, you'll see numerous phones with various features and price points coming to market. Because the phones are new, it will still require debugging new issues, even though the phones are more powerful."

On top of that, developers must still support the phones that were on the market last year. While phones that were on the market last year have been discontinued, "you have a legacy of a few million handsets that you can't ignore, if you're trying to make money," says Matthews.

Fortunately the industry has been producing a number of standards bodies from basic handset feature sets to 3D standards to help guide the handsets in a common direction. Unfortunately, with number portability, the carriers are now on the make for things to differentiate themselves from the competition. And the handset manufacturers are always in a horse race to see who has the coolest iron on the block.

As Matthews says, "One possible direction, for which there are positive signs, is for carriers to start emulating the DoCoMo model, where the handset vendors have to conform to a base hardware platform. That's one logical direction for handsets to go—until then, it's going to be a wild ride."

Until that point, the chaos actually works to open the world for more developers. The Hot list and decks for each mobile phone model are customized to only show the games that work with that device. That means there are more places to get a toe in. And given the growth of the number of users with handsets capable of playing games and the success of the micro-payments, extending that toe is clearly worth it.

Optimizing the Content Pipeline

After years of being almost completely technology-driven, the driving force behind games is finally swinging towards the game content itself. Major technological leaps are not making enough of a difference to set games apart from each other. Game content will also continue getting larger and more complex, as the amount of it going into AAA games doubles every few years. Yet, since there's no hardware upgrade for the artists and designers themselves, content doesn't get created any faster.

A successful game needs to provide top-notch content, and the best way to provide it is to optimize the content pipeline so artists and designers can create, preview, add, and tweak new content as easily and rapidly as possible.

The content pipeline is the path all the game assets follow, from conception until they can be loaded in the game. Game assets include everything that is not code: models, textures, materials, sounds, animations, cinematics, scripts, and so forth. During their trip through the pipeline, assets might be converted, optimized, chopped to bits, or combined, but come out in the format that will be shipped with the final version of the game.

The first issue to consider when defining the content pipeline is its efficiency. With large teams of artists and designers constantly creating and tweaking game content, the content pipeline becomes a critical path. A slight inefficiency in the pipeline, such as taking one full minute from the time a change is made to the time it can be seen in the game, can easily cost a company thousands of wasted man-hours during the course of a project. Alternatively, if the content creators don't preview their work as frequently, the overall quality of the game will suffer.

The other main point to consider is robustness. The content pipeline is the jugular vein of a project: if it breaks it can quickly kill the whole project. You can't afford to have 30 idle people waiting for the pipeline to be fixed, or working around it and consequently losing half their work. Whatever happens, the pipeline must always work correctly.

Bird's-Eye View of the Pipeline

What does an asset pipeline look like? It depends on the project. On one extreme, in some projects the pipeline is minimal and informal: assets are exported from their tool and loaded directly in the game. While that might be sufficient for small games, it usually doesn't hold up well in large projects. Where are the files stored so multiple people can work on them? How are assets for multiple platforms dealt with? How can the format of the resources be changed easily? How can any extra processing be applied to them?

On the other end of the spectrum, pipelines can be very deep and elaborate. Adding a new asset to the game requires going to the pipeline guru and asking

NOEL LLOPIS | *Noel is a software engineer at Day 1 Studios, where he developed the technology for the MechAssault games. He's also the author of* C++ for Game Programmers. *Contact him at nllopis@gdmag.com.*

MechAssault 2's models were exported as XML files including hierarchy information, meshes, and vertex data.

him to add the new content, causing turnaround time to suffer significantly.

This article presents a general pipeline that many different game projects can adopt and modify to fit their needs. It is fairly lightweight and provides a quick turnaround time, yet it allows for any number of expensive steps to be performed on the assets along the way. This is the pipeline that we're using at Day 1 Studios for our current Xbox project, MechAssault 2. A previous incarnation of this pipeline was used in the first MechAssault. This pipeline might be a good starting place if you're just beginning a new project, or you can try to adapt some of the ideas that make sense for your current situation.

Figure 1 shows the pipeline for some of the major assets in MechAssault 2. The following sections describe in detail each of the pipeline stages.

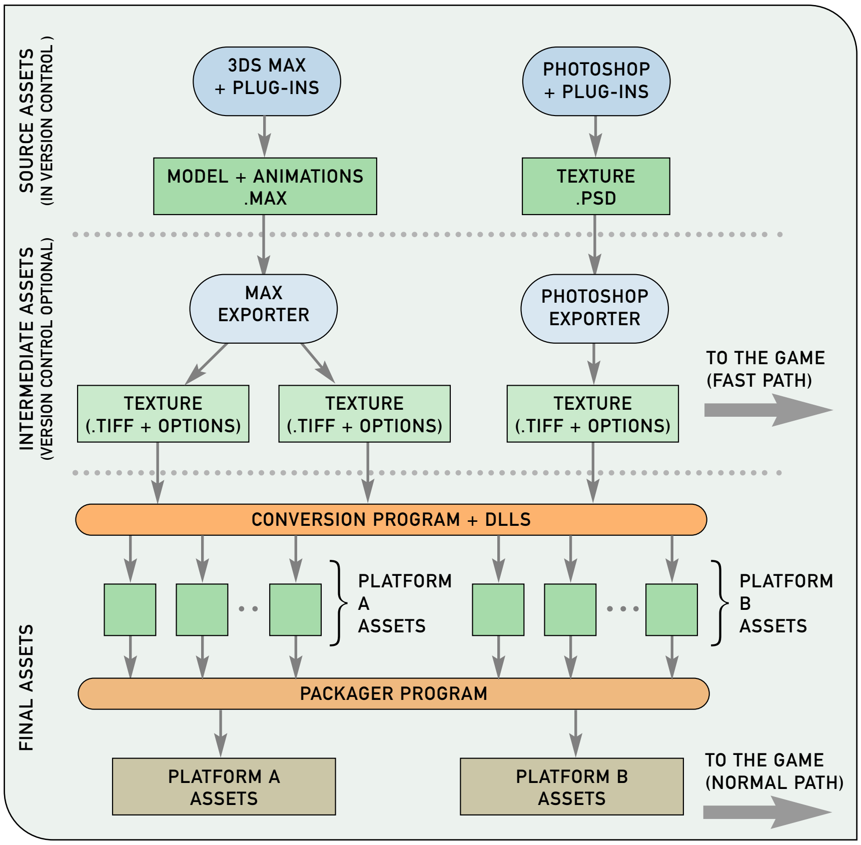


Figure 1. A partial view of the MechAssault 2 content pipeline (only models and textures shown).

Source Assets

Source assets are those created by artists and designers, usually through some specialized tool (both in-house tools and off-the-shelf ones). A source asset is one that can be put into the pipeline and will be converted into the final asset without any human intervention. The key idea is that source assets should contain all the information necessary to add them to the game correctly. A material should have all its specular parameters specified, and a model should have all the flags in the weapon barrels so the game knows from what point to shoot projectiles. This is what will allow us to automate the pipeline later on.

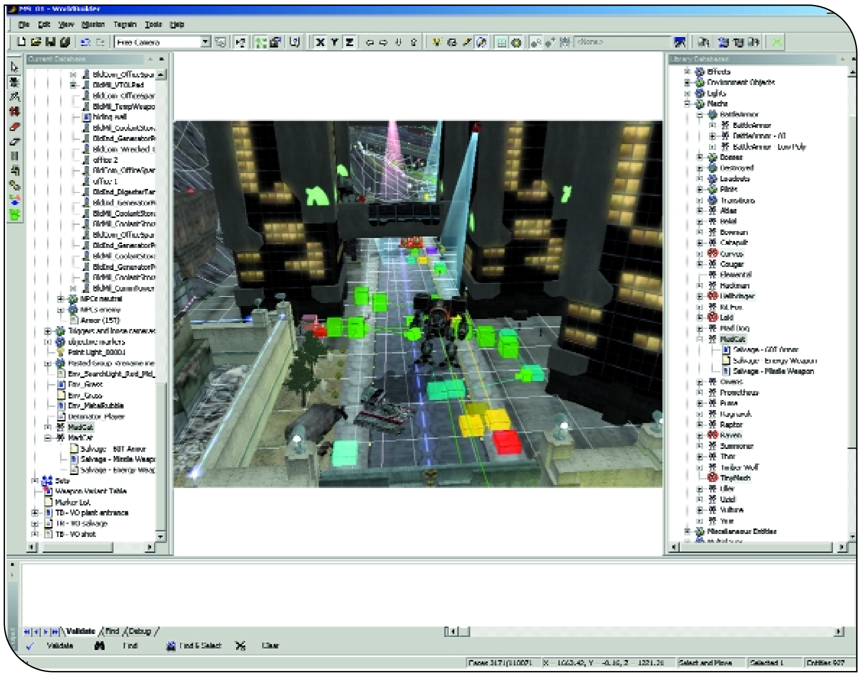
The way source assets are created can vary significantly. Sometimes they are created through a set of different, very specialized tools (one for modeling, one for texture creation, one for specifying game information, one to lay out a level, and so forth). Other times, one large tool (often done as a plug-in in the model-editor program) can be used to create all the assets and export full levels.

Since the source assets contain all the information needed for the final assets in the game, we should treat them very carefully and protect them from being lost or accidentally overwritten. Using a version control program provides a centralized location for all assets, prevents people from overwriting each other's work, and keeps a history of previous versions of each asset. Keep in mind that source asset files can sometimes be as large as several hundred megabytes each, so make sure your version control program is up to it and can also deal well with binary files.

Pre-rendered movies and sound are also game assets, but they're often treated differently because of their huge size. They might have a slightly different path through the content pipeline: maybe movies won't be kept under version control, or maybe they will but only the most recent version will be kept in the database.

Intermediate Assets

Intermediate assets are exported directly from the source assets, usually with the tool they were created with. This requires a bit more plug-in work in the form of exporters if you're dealing with off-the-shelf tools.


Manipulating MechAssault 2's assets in WorldBuilder.

The intermediate assets are in a format that is very easy to read, parse, and extend without breaking backwards compatibility. These assets should contain all the information we could possibly want in the future, even if some of it gets discarded before the end. Loading performance is of no consequence at this point; we'll leave that up to the final assets.

Plain text files are the perfect match for our requirements. In addition, XML is a particularly attractive option to structure those text files, especially if we need to represent information hierarchically. Doing so will also provide us with a whole range of tools and APIs to edit, parse, and transform files with minimal effort. If XML files are overkill for your needs, you can still use a simpler format like INI files or even write your own with minimal effort.

Why have this intermediate asset format? The main reason is to provide a buffer between the source assets and the final ones. Final assets are optimized to load blazingly fast, but as a result, their format will often change and render previous versions unusable. In an ideal world, we would be able to efficiently re-export all source assets automatically into the new final asset format. Unfortunately, we don't live in an ideal world, and that is often impractical. Many off-the-shelf tools used for modeling and texture creation are not easily and efficiently driven from the command line to batch re-export thousands of models at the time. The intermediate format is not likely to change much during the course of a project, so we can always use it as a starting point to generate our final assets.

Another reason for having this intermediate format is to defer some of the complex and time-consuming operations like mesh optimizations or lightmap generation until a later time instead of doing them at export time. Also, we often end up creating a set of final assets for each platform we're dealing with. Without the intermediate format, artists would have to export a set of assets for each platform, ensure they're in synch with each other, and wait while each set of assets is converted and optimized every time an asset is changed.

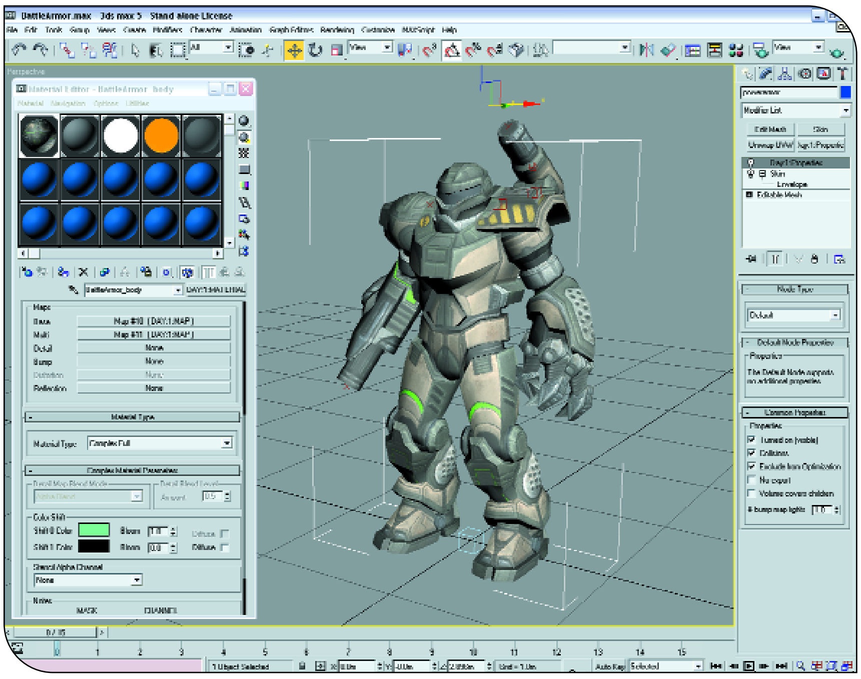
Last but not least, having an intermediate format provides an excellent point for debugging and experimenting for the programmers. Intermediate assets should be in a very easily readable format, so anybody can view the contents of the asset, and even make small modifications for testing and debugging purposes without having to re-export them.

For MECHASSAULT 2 we're exporting full models into one XML file: all the hierarchy information, meshes, vertex data, and materials are included into one large XML file. One of those files for a detailed model can easily be 3 to 4MB.

As much as plain text is a really nice format to work with, exporting textures into text format would be overkill. In the case of textures, we export them into 32-bit TIFF files with custom information in the comments field containing any information the artists specified from within Photoshop: bit-depth, dithering, mip-mapping options, and such. The advantage of this format is that all the image is still there since it's exported at full 32-bits (although soon we'll have to worry about larger color channels), and images can be examined with any program that displays TIFF files. Again, mip-mapping, changing bit depths, and dithering are not cheap operations we want to do at load time, but we'll postpone all that until later.

It can be very tempting to modify an intermediate asset directly for many reasons: a quick change right before a milestone, or the change we're trying to make is not exposed through the plug-in of the tool, for example. We did exactly that in the first MECHASSAULT and ended up regretting it. We were too pressed for time to provide good plug-ins for our custom material types, so artists would modify material parameters on the intermediate assets directly. Those parameters would get overwritten the next time somebody else re-exported the model and would have to be re-entered by hand. If you absolutely must modify intermediate assets, then consider providing full re-importing capabilities back into your source assets; however, this is usually not a trivial task. For MECHASSAULT 2 we're sticking to only making changes in the source and providing much better plug-in support, and things are running much more smoothly.

Even though this intermediate format is very flexible, chances are there will come a time during a project when it will be necessary to break backwards compatibility. Don't fight it; get ready for it instead. Make sure you have versioning as part of the format, and that you can easily run a script through all exported intermediate assets and convert them to the new format. Since you hopefully chose an easily parsed format such as XML, the conversion process should be almost painless.



Custom material editor and node properties plug-ins for 3DS Max.

Final Assets

The final assets have been highly optimized so they can be loaded and used as efficiently as possible in their target platform. The specific file format doesn't have to be particularly robust or withstand many format changes since the final assets will be regenerated many times a day. The number one goal here is speed. Ideally, this resource should be a direct memory image of the format it'll be in when it's loaded in the game, so that it can be loaded straight without any parsing. The standard warnings about optimizations also apply here: don't blindly optimize everything, and spend your time in those assets where you'll get the most benefit.

If you're doing multiplatform development, you'll probably want to have one set of final assets for every platform. That way Xbox textures can be in their own format, and PS2 textures can be packed and formatted differently. At Day 1 Studios, even though we were only developing for the Xbox, we had two sets of resources: one for the Xbox, used in the game, and one for the PC, used in our PC-based tools.

The type of operations done at this stage range from quick ones such as simple format changes, to somewhat expensive ones such as mip-map generation, dithering, or compression, to really time-consuming ones such as lightmap generation and mesh optimizations. At this point we don't care too much about how long it takes to perform an operation; we just care about the final result and how optimized the final asset is. As a result of this step some assets could be split into different assets (for example, our model XML file is split into several smaller files containing vertex and index data), and sometimes multiple assets get combined into one larger asset (packing a set of textures into one larger texture).

At Day 1 Studios, this step is performed by a command-line conversion tool, which takes a set of intermediate assets and produces the final assets. The actual conversion happens through a set of conversion plug-ins implemented as DLLs, each of which takes care of converting a particular type of resource (identified by file name extension or a header on the file itself). Any resources not handled by the conversion plug-ins are just copied straight through without any modifications.

Be warned that doing a full resource conversion on all the assets of the game can be a very time-consuming operation. There can easily be hundreds of thousands of files, and each of them needs to be loaded, parsed, converted, and saved in the new format. This process can take up to several hours even with a fast CPU and a very fast hard drive. Minimizing this time will help with the overall turnaround time, so you might want to profile the conversion program to find any obvious bottlenecks. Other solutions include doing incremental conversions (only convert files that have changed since the last build), and doing distributed builds (have a farm of machines where each one takes care of converting a subset of the resources).

During this step we should also generate errors and warning messages as necessary. Sometimes an asset won't meet the requirements to be converted successfully (for example, a texture might be marked as needing mip-maps but it has dimensions that are not a power of two). It's much better to prevent invalid assets from making it this far down the pipeline by preventing them from being generated in the first place, but we should still be ready for them here. By keeping a log of all the failed conversions and all the warning messages, the art and design leads can quickly identify what the problems are and correct them for the next build.

If you have specific dependencies between resources that need to be enforced (for example, packing all the textures for a particular model together and the textures need to be converted before they're packed), you might want to use an existing dependency-management tool such as Make or Jam.

Catalog Files

The final step of the pipeline consists of packing all the loose final asset files into larger catalog files. A catalog file is nothing more than a large file containing other files inside, possibly along with their names and hierarchy information.

The advantages of using catalog files are pretty obvious: reduced number of open/close file operations, enforced physical proximity between files, easier and more efficient distribution of assets, faster directory parsing, and so on. Surprisingly, not all modern games provide catalog files with their assets (you can usually tell who the guilty parties are by the outrageously long load or install times).

Using a standard catalog file format (such as .ZIP, .CAB, or even .WAD) is very convenient because there are already-made tools to create, load, and view them. On the other hand, you might not have as much control over them as you want (for example, to enforce a specific file alignment or ordering for optimal seeking performance on a DVD).

In this step, the final assets for each platform are packaged into separate files. You might want to have separate catalog files for each level, or have sound and music on a separate catalog file from the rest of the assets. Those strategies will depend on how you need to load resources in your game. Once these catalog files are created, they can be copied onto the network and everybody can copy them locally to their machines or work with them directly from the network if there's enough bandwidth. The game and the tools should be able to read these files directly and load any of the resources inside transparently.

The Fast Path

Unfortunately, the process we've described so far isn't exactly speedy. To make a change and see it in the game, we need to go through the following steps: modify a source asset, export the intermediate one, check it in version control, kick off a resource conversion, and wait for a few hours until new catalog files are ready. That doesn't meet our earlier requirement of a fast turnaround time.

The solution is to provide a fast path into the tools and the game that bypasses all the time-consuming steps. In our case, we made it so both the tools and the game could load intermediate assets as well as the final assets (see Figure 1 again). All an artist has to do to see how his updated texture looks in the game is to export the intermediate asset, copy it on the directory where the game or tool expects to find it, and run the game. You can even provide a macro or a button that does all those steps at once, including launching the game to the correct level. The artists just make a change, push a button, and see their new assets in the game. It can't get much easier than that.

How exactly is the local file loaded instead of the one residing in the catalog? That's part of the magic of the file manager. As far as the tools and the game are concerned, they're just opening a file like any other. However, our file manager gives local files priority over catalog files so they override any similar entries in a catalog file. Whenever the game attempts to open that file, the file manager redirects it to the local file. Otherwise it loads it from the catalog as usual.

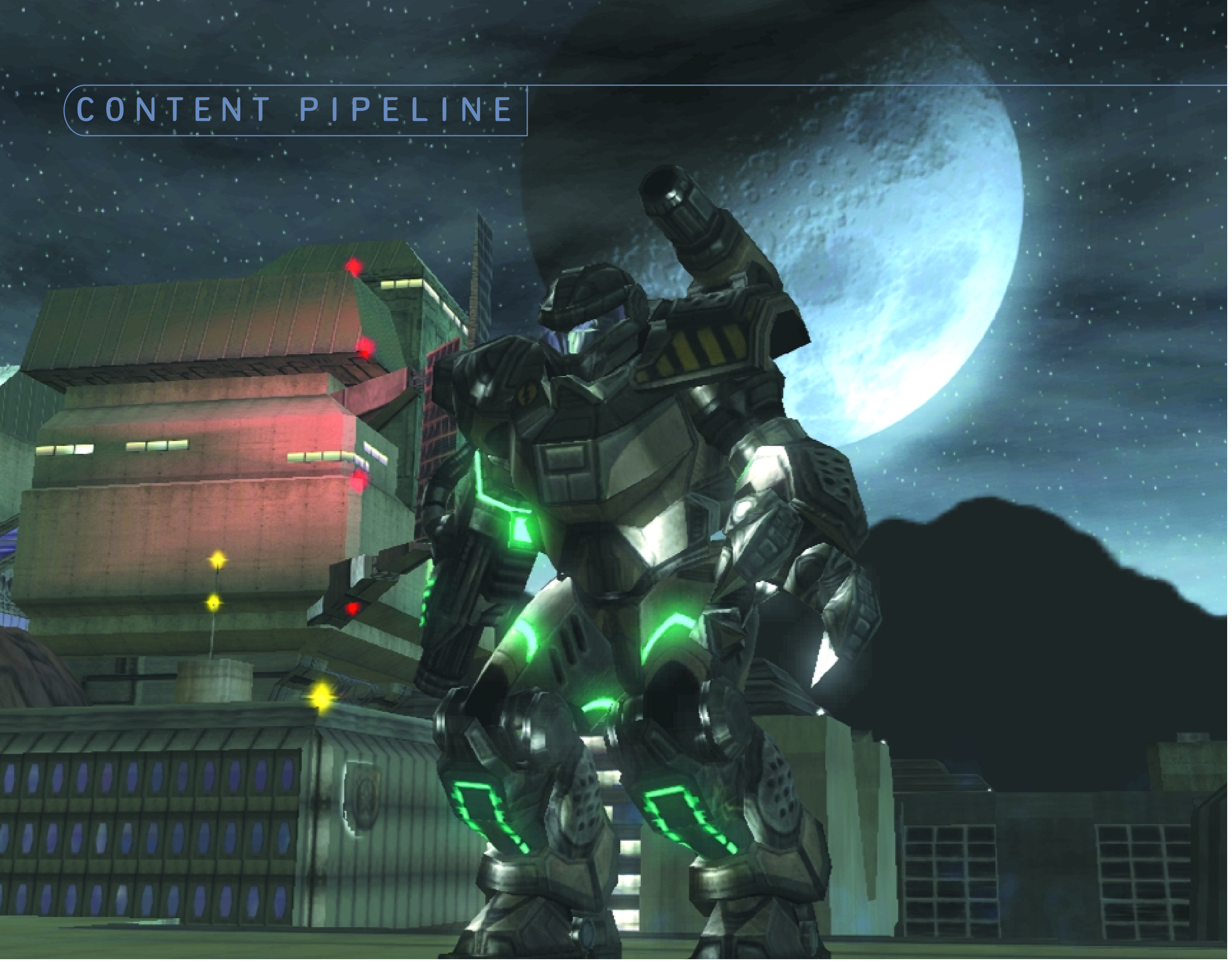
Since we designed the intermediate format so it was easy to parse and modify, but not fast to load, isn't that going to slow loading levels down to a crawl? It would if all the resources in the level were in the intermediate format. The idea is that only the assets that artists and designers are working with at the moment are going to be found locally in the intermediate format, so load times are only going to increase by a very small amount.

This approach also requires that the game and all the tools have code in them to load and parse assets in their intermediate formats. That is not a trivial amount of code, but chances are you've already written it to make it part of your conversion tool. However, you might want to strip this code out of the final shipped game, so separate it cleanly and surround it with conditional compilation statements so it's easy to remove.

There is one important lesson we learned from using this fast load path: make sure that your fast load path and your normal path (following the rest of the pipeline) both produce the same results; otherwise you'll have some very disconcerted artists who are not sure why their assets look different after they check them in. In our case, we were using one library for texture conversion in the slow path and a different one in the fast path. The results were often similar, but sometimes different enough to raise a few questions. Whenever possible, use the same code in both paths.

Putting It All Together

Now that the whole pipeline is in place, we can finally start thinking about automating the process. It should be relatively straightforward to create a script to get the latest source or intermediate assets from source control, run the conversion process on all of them, and package them on the catalog files. Perl and Python are some good glue languages for those types of tasks. The most difficult aspect of the script is making it robust enough to deal with errors (network going down, version control server being down, and so forth), so take good care of that from the beginning. With such a script in



place, we can run the resource build automatically once or twice a day, or on demand when new resources are needed.

The more we automate the content pipeline, the more important feedback becomes. People are not going to be watching every step of the pipeline, so we need to collect all the important information and deliver it to the people who care about it. In addition to gathering all errors and warnings, we might also want to collect other information, such as memory footprints, texture usage, or even some rough performance statistics. All that is best done as a final step to the resource build, running each of the levels in the game with the latest resource and executables. As a side benefit, it also serves as a very rough smoke test of the build.

Robustness was one of the goals of the pipeline from the very beginning. Part of it involves making sure the conversion and packaging tools work flawlessly and report any errors correctly. The other part is making sure the game and the tools are never left in an unusable state because of bad resources. A good philosophy to maintain is that bad data should never break the game or tools; an artist or designer should never be able to crash the game. It might sound a bit radical, but it's worth aiming for that goal. Any engineering time spent towards this will be paid back many times over as soon as assets start being added to the game at full speed. When loading a level, take the time to report any loading or initialization errors, disable the entities that had problems, and move on. In addition to that, it's helpful to put some sort of ugly debugging model (a big pink lollipop in our case) in place of any entity that failed initialization.

Further Work on the Pipeline

Turnaround time is very good already, but we'd like to make it even shorter. We'd like the game to detect that some assets have changed and load them on the fly. This can be particularly beneficial for games without discrete levels where reaching specific locations is a time-consuming task.

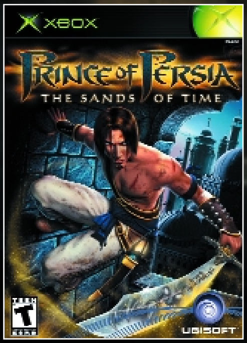
A full resource build for MECHASSAULT 2 can take up to an hour and a half. We'd like to further investigate the possibility of doing distributed builds. There are open-source, general frameworks for doing distributed operations that could be easily added to the process. We might also want to look into integration with the build systems such as Apache Ant for added robustness.

Games are very different from each other and teams are organized differently, so content pipelines will vary significantly from project to project. It is important to identify the assets for a given game, what kind of operations will be done to them, who will be working on them, and at what stage of the development they will occur. Use whatever pipeline organization works best for your particular needs and automate as much of it as possible. The artists and designers in your team will thank you for it, and you'll end up with a much better game in the end.

Return

POSTMORTEM *yannis mallat*

GAME DATA



PUBLISHER:	Ubisoft
DEVELOPER:	Ubisoft Montreal
NUMBER OF DEVELOPERS:	65 at peak (excluding testers)
LENGTH OF DEVELOPMENT:	27 months
DEVELOPMENT HARDWARE:	Average machine: Dual AMD Athlon 2000, 1GB RAM, Windows 2000, PlayStation 2, Xbox, and GameCube development kits, PlayStation 2 Performer Analyzer
DEVELOPMENT SOFTWARE:	Microsoft Visual .Net 2003, Metrowerks CodeWarrior, PlayStation 2 Tuner, Incredibuild
RELEASE DATE:	November 2003
TARGET PLATFORMS:	Xbox, PC, GameCube, PlayStation 2, Game Boy Advance
PROJECT SIZE:	4188 files, 1,263,580 lines of code
PS2 BUGS:	11,520
XBOX BUGS:	936
GAMECUBE BUGS:	1,004
PC BUGS:	1,072
TOTAL BUGS:	14,613

YANNIS MALLAT | *A producer for Ubisoft, Yannis's primary titles include* PRINCE OF PERSIA: THE SANDS OF TIME *and* RAYMAN ADVANCE. *He also worked on production for the movies* Little Nicky, The Emperor's New Groove, *and* Dinosaur.

Ubisoft's Prince of Persia: The Sands of Time

Prince of Persia, an original creation by Jordan Mechner, was first released in the U.S. in 1989. The game, which follows the adventures of a young prince's efforts to save a princess, is regarded by many analysts as the first true action/adventure game. The Prince of Persia franchise has seen two sequels since its conception: Prince of Persia: The Shadow and the Flame (1993) and Prince of Persia: 3D (1999). By 2001, Ubisoft felt the time had come for the return of the prince.

When Prince of Persia was first released in 1989, it got the attention of the game industry. It became an instant classic and laid the foundation for the action/adventure genre. The settings were strong, the storytelling was compelling, and the animations were groundbreaking. The game established new standards for what the public should and would expect from videogames to come.

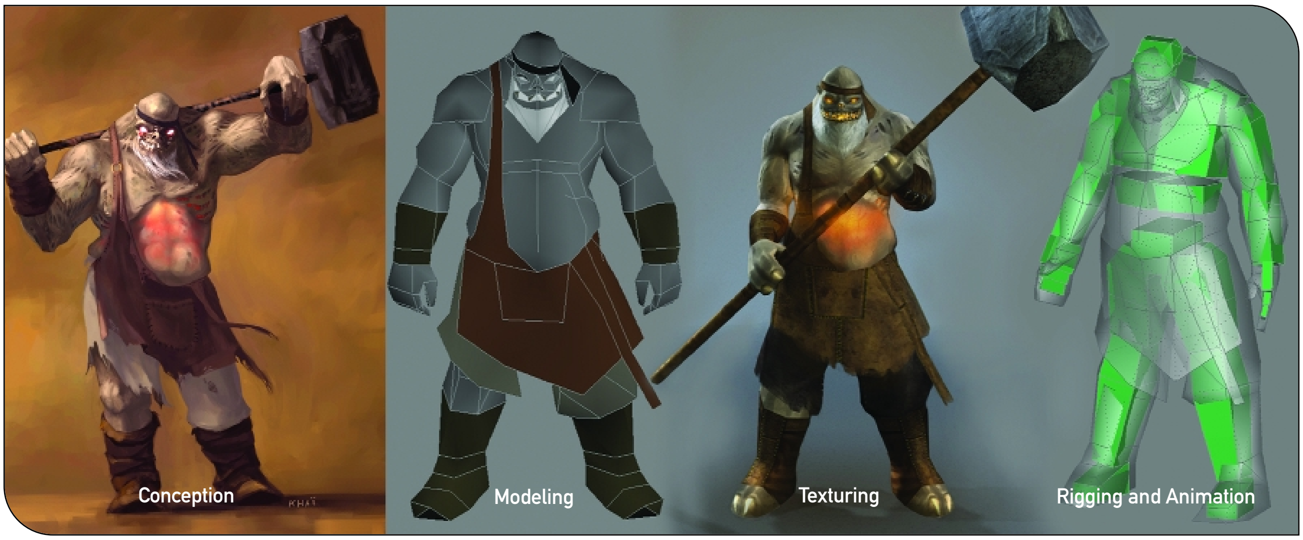
By May 2001, a number of platformers had been released since the launch of the original Prince of Persia. Most of them were inspired by at least some of the elements that made Prince of Persia an important achievement. In Spring 2001, Ubisoft announced it had acquired the Prince of Persia license and gave the Montreal team a mandate to start the conceptual phase of the project.

Early on we identified the three core areas that made the original game a success. They are 1) captivating animations and character movements, 2) intense fight sequences, and 3) clever and challenging levels and the gameplay built around them. They were the essence of the brand and, if used with the right formula, the universal ingredients for a stellar action/adventure game. We considered them the heart and soul of the project.

So, there we were, a team of seven, laying down the basis of what would later become Prince of Persia: The Sands of Time. Two game designers worked on defining the main concept, helping to build prototypes in real time with the technical team. One animator created the major moves that essentially brought the prince to life.

We then integrated two engineers into the process. They started the engine studies and helped the design team conduct gameplay tests. A concept artist was added to the mix to illustrate game design ideas and provide initial art direction (to the extent possible at this stage). He also contributed creative ideas. The final piece of the puzzle was the producer, someone who would also act as a game designer and creative consultant, a role I gladly accepted.

A couple of months later, when we were able to present our first mock ups (AVI files showing how the prince could move and interact with his environment), we asked the original Prince of Persia creator Jordan Mechner to look at what we had done. The result of the first presentation was inspiring. He was duly impressed. He hopped on the train and the core team started chugging along full steam, beginning with the pre-production phase and then switching to the production period.



Character production workflow, showing one of the prince's enemies from concept to game.

What Went Wrong

1. Late arrival of the artistic director. While the project effectively began in June 2001 with a fast-track conceptual phase, the art director, Raphael Lacoste, did not join the project until late April 2002. Although it didn't impair the final art direction, the very late arrival of our artistic director did create a huge challenge in time management for the team of artists.

Prior to his arrival, several prototypes had already been made showing the prince's movement set, level design ingredients, and some technological breakthroughs, but nothing very impressive. There was almost no art at all. The game's potential was demonstrated with some very basic level design blocks and monochrome textures.

Raphael's first task was to define the artistic direction and style of the game and to develop all the necessary tools. Light maps were to be added to the engine at the 11th hour of pre-production, along with many other effects (volumetric fog, filter, glow, and so on). The most difficult challenge for the modelers was to keep a steady production pace for the maps while learning about upcoming and unfinished tools. As a matter of fact, the first final art wasn't available until the E3 2003 demo.

Coming back from the show, the team saw the demo as the standard of quality that should be consistently present throughout the whole game. This seemed impossible, considering just how much we still needed to produce. The demo was approximately 1/30 of the whole game. But the risk management output (including some scope reduction) and the tremendous efforts of our highly motivated team resulted in visual quality that surpassed that of the demo.

2. Fuzzy validation process. From the beginning, we knew that dealing with such a well-known license would present some challenges. We needed a huge pre-production process to help us establish clear goals, which included completing character behavior, macro designs, a compelling storyline, and all tools. A playable proof would then allow us to move forward into production.

That said, we didn't think pre-production would last as long as it actually did. When level production began, we had planned for 10 months; it eventually took more than 14, with a good list of tools and fighting behavior still in pre-production. Maintaining the right balance between creation and production was hard, and there was no clear distinction between what was approved and what still needed improvement.

The prince's behaviors were often changed, refined, and tweaked, which required major modifications each time. All of this was good for the game's overall quality, but we had already lost precious production time designing, implementing, and rejecting several complete fight systems (in animation and AI). The result was a chain reaction that put other important deliverables in jeopardy. For instance, we started Farah's (the princess) AI development later than expected. We didn't have enough time to really polish the generic AI-supporting level-design scripted events. We had to take care of cooperative gameplay case by case, level by level, situation by situation. All this postponed the start of the real debugging period. We were faced with a mountain of bugs that had to be fixed. But the gold master release date was not going to budge.

3. Complicated enemies. The prince's character was the subject of intense work during pre-production.

With more than 780 animations, he was obviously the most significant—and the largest—component of the game. Unfortunately, this left less time and fewer resources to develop those who would allow him to exploit all his abilities: his enemies.

Enemies represent particular level design ingredients. Being extremely dynamic, they need to complement the main character's combat skills. At the same time, they should also increasingly challenge the players and surprise them with unexpected behaviors in any given situation. We also used specific enemies as tools to teach the player how to fight better—an instrumental aid in the players' learning process.

Due to the late delivery of final maps, all the enemies' behaviors had to be developed and coded on placeholder maps (basically a floor), which did not take into account the geometry of the actual maps. Obviously, in this situation, the enemies' AI came out way too bland, compared to what it should have been. Contextual enemies (such as the Sandbirds, Sandtigers and other mythical creatures) were extremely cost-inefficient to produce. Some of them simply had to be cut, whereas all the bipedal enemies later required a significant debug process.

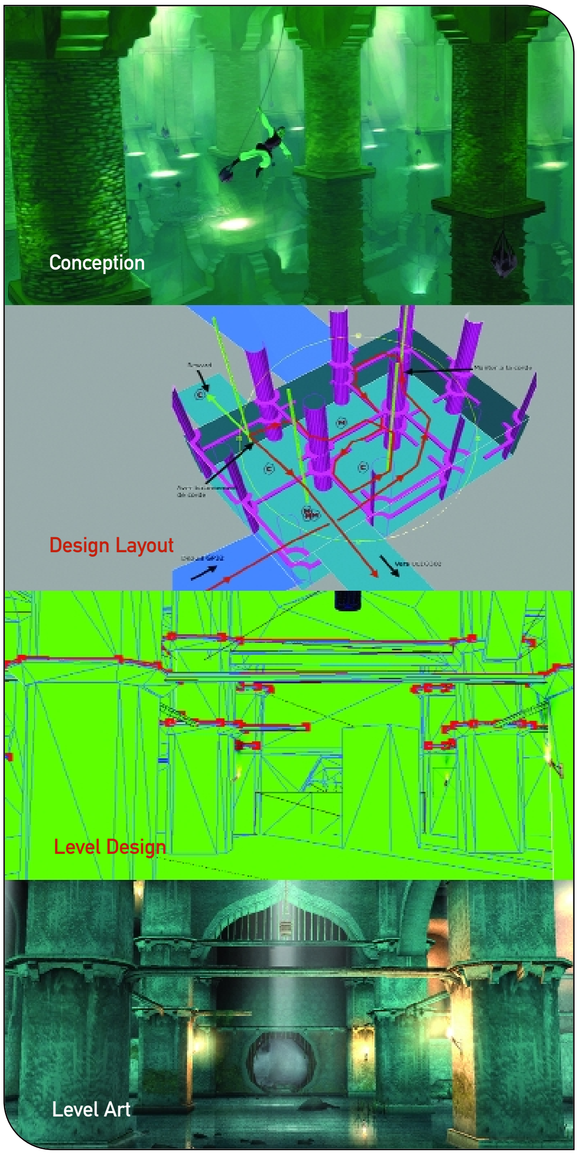
4. Lack of strong technical level design. From the beginning, our game was all about level design. Each of the prince's moves drove the micro-gameplay. Much of what the players would enjoy was rooted in level design. Every aspect of the prince's behavior or animation had a match in the geometry of a level. The game was very context-sensitive: you need a wall to make a wall-running maneuver; you need a column to slide down it.

We had to make our technical features behave flawlessly. First of all, the dynamic loading was not ready right from the start of production, so we had nightmares getting everything to fit in memory and adjusting pre-fetch settings. Making all these adjustments was very tricky because we wanted everything loaded in time. To avoid sudden movements or pop-ups, we had to make everything highly interdependent. On top of that, we had to make sure our rewind feature was always working, since this was how objects/enemies were destroyed—through dynamic loading portals and the like. Combine all these with a bunch of eager QA testers and you get a pretty intimidating bug database. Thankfully, the level designers and the programming team were able to squash all of the bugs.

So, were we starting to see light at the end of the tunnel at this point? Well, not quite. We were forgetting another source of problems. The game wasn't crashing anymore, but the enemies were forgetting their objectives. This led to broken gameplay, where enemies no longer saw the prince or attacked him. Furthermore, since you couldn't beat them, you couldn't complete the level. Even worse, the princess was completely forgetting many of her crucial goals.

So much could have been done in the earlier stages of development to prevent these problems. If only the maps and gameplay had been delivered in advance, a dedicated technical level designer could have foreseen all these issues and fixed them before alpha. Once again, we were not dealing with the problems in a strategic way; we were putting out fires as they occurred. Meanwhile, we were creating a mountain of bugs to deal with later.

5. Data control. The way the engine was built meant the game data was stored in one master file that contained everything for the developers to review: maps, models, AI, and the rest. Everything was centralized in this file except sound and videos. The situation didn't allow for multiple concurrent data access on the same file, at



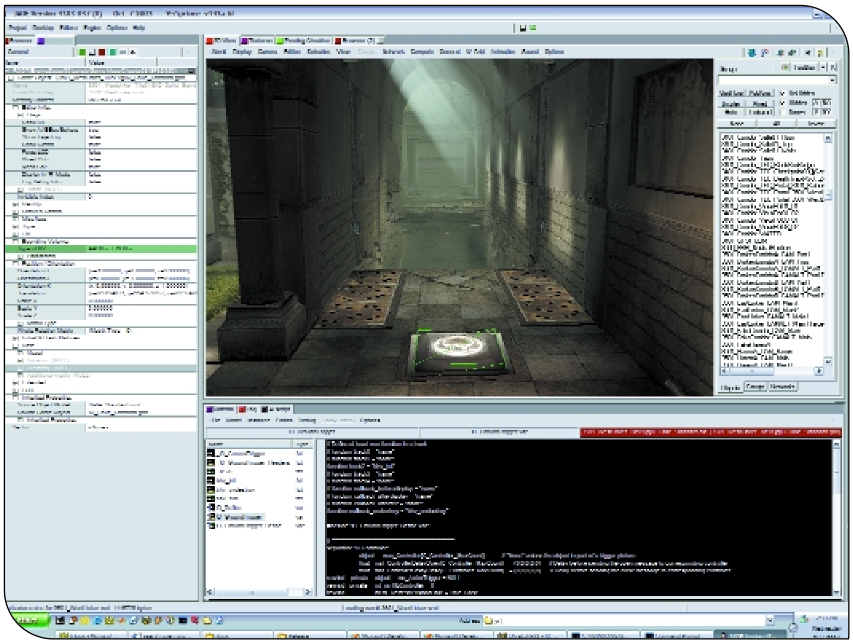
Map production workflow for building the Maharajah's palace.

least without written permission.

We soon realized our team had become too large to allow everyone access to the master files at the same time. We inherited a system that was designed with a small team in mind, but it didn't scale well to an army of 45 in crunch mode. Many problems occurred: data was overwritten; the server crashed; files got corrupt. A lot of time was wasted because people had to wait for their turns to enter their changes on the network.

We tried to optimize the data control at the very end of the project, by building a "data monkey" solution that would allow simultaneous access through a server while maintaining a single repository for game data. Unfortunately, the attempt to build such a tool came too late and we never had the chance to alter the system. The risks involved were too serious.

One little thing we did, however, was set up a simple file server to manage the timing of all check-ins. At least the developers could work on something else locally while waiting for project updates, and we could give priority to people trying to make critical changes.


Lighting sets the mood in a corridor of the Maharajah's palace, as viewed through Ubisoft's internal playable level editor.

What Went Right

1. The will to achieve. A major element that contributed to the success of the whole project came from the team itself, and we managed to keep the initial motivation and chemistry strong right up to the end. The team was (and still is) a collection of extremely talented people in every field.

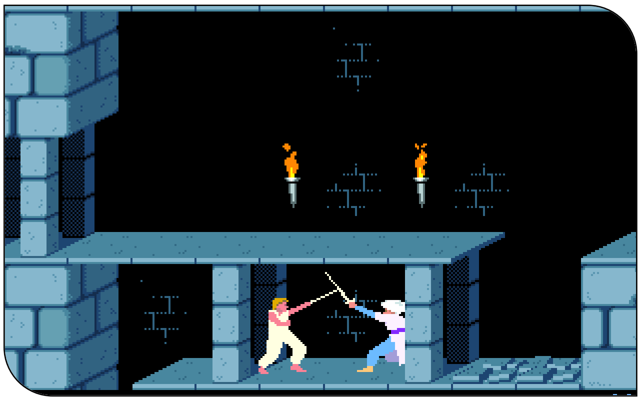
The project started well with a very powerful initial deliverable that helped everyone to clearly see what we were aiming for. At the start, the team was composed of less than 10 core people in complete harmony with one another—a tight-knit family. We were able to maintain the most effective form of communication: honesty. Speaking harshly about things that needed to be discussed was not a problem; we shared a common desire: the success of the project. No ego trip threatened the team's interest. Integration of newcomers could have disrupted this cohesion, but it didn't, because we didn't add large numbers to the team all at once. Instead, we chose to incorporate newcomers one at a time, easing them into the unit gently.

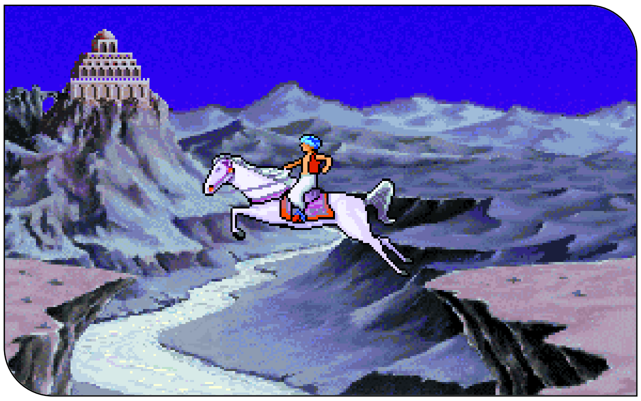
A succession of morale-boosting events helped maintain the highest level of energy and motivation within the team: Sony decided to show the game at its E3 booth, and our own demo of the game at E3 was well received: people turned out in droves as word spread quickly that this was the game to see. Our high motivation level and confidence in the project allowed us to deal with an incredible amount of pressure (time and quality, for starters), accept some difficult realities (scope reduction and so on), and work extremely hard for a very long period. From the E3 demo preparation (late February) to the very end, we worked on average 16 hours per day, peaking at 20 to 48 consecutive hours sometimes. It's not a good model and we would prefer not to work like this again, but it was essential and the whole team was up for it, with absolutely no complaint.

2. Synchronization between animation and AI. The prince, as he appears in the final game, was our very first success and could not have been achieved without a fantastic duo that was paired up at the very beginning. The lead animator and lead AI for the main character worked very closely together. There was no question of a separate animation production on which we would simply map the AI afterwards. Both were conceived together, created together, and generated and implemented together. The two guys actually placed their desks side by side and worked as if they shared one brain. This is apparent in the way animation and control (AI) work seamlessly together in the final version.

3. Risk management. When tough decisions needed to be made, we made them. We reduced the scope of the game at two crucial times: just before Christmas 2002 and right after E3 2003. Fortunately, these decisions were made early enough in the development process.

The first scope reduction was the





The original PRINCE OF PERSIA titles stretched the boundaries of animation and art design.

hardest to make, because we were still far enough from the gold master date to convince ourselves "everything would be just fine." Specifically, we were talking about cutting an entire chapter that took place in a slave village featuring exotic gameplay elements. Cutting this specific chapter meant having to tell the story very quickly. We accepted this decision because, in the end, everyone agreed it was the right move; if we had made this decision later, or worse, if we had refused to trim it, we would never have been able to finish the game on time.

When we got back from E3, we faced the bitter reality of chaotic production: most maps were not running at all; some were not even close to completion. Thus, the second scope reduction was logistically easier to make, but still hard on the team: it meant cutting some things that we had spent a lot of time working on, stuff that we were proud of. But, here again, if we had made this decision even a week later, we wouldn't have met our deadlines.

4. Playable editor and other tools. As I've said, this game was all about level design. In PRINCE OF PERSIA: THE SANDS OF TIME, the gameplay was created mainly by the environment. Technically, all the level design distances had to be perfectly adjusted, because the gameplay could not exist with any degree of approximation.

When the prince grabs an edge from a vertical-wall rebound, his detection zone should be perfectly in synch with the edge (in terms of spacing). This could have been a very strict limitation in level design creativity, but it wasn't.

The editor was built to let level designers play with a 3D view. This allowed for quick corrections, thanks to a trial-and-error approach. Adjusting a column, adding a rope, or removing any level design ingredients were done on the fly and tested immediately by the level designers. The most interesting and crazy level design sequences were created in a very short amount of time. When the map was on the modeling side, it was also extremely useful to check whether the gameplay was altered by the addition of extra art geometry (such as a light torch on a wall where the prince needs to run). The tool helped us quickly devise interesting gameplay ideas during pre-production, then produce art geometry without wasting time compiling everything for a look at how the map was played.

5. Integrated testing. Finally, we provided development kits to as many testers as possible. At peak time, we had 14 PlayStation 2 development kits for the team, four of which were solely dedicated to QA testers reproducing very rare crash-bugs (with a special debug "strike-team" to take over the machine with a debugger and reverse-engineer strange bugs in retail code).

This started a creative solution to a recurring problem. One day, we realized one of our testers was great at finding A bugs—the rare, nasty ones. She was able to find bugs no one else could. Initially, each developer who was assigned to her bugs got frustrated due to the time commitment in fixing them. Then, we asked her to join the team, equipped her with a development kit, and with her working on the game itself, we got our A bugs curve back to normal. We replicated the model to up to four integrated testers within the team. This dramatically accelerated the pace of finding and fixing bugs, freeing some time for the developers to focus on the fixing side. Eventually, these testers got into the groove of things and spent many long days and nights contributing to our collective masterpiece.

1,001 Nights Later

And there we were, at the end of October 2003. After all the crazy events we had experienced in the previous 36 months, the gold master was finally delivered and the CD-ROMs were pressed. We couldn't believe it. We had made it.

This team can be very proud of what it achieved. I would gladly work with all of them again in a second (in fact, I am), and we are now ready to welcome newcomers for the next installment of our adventures.

Return

Preserving Your Games

Illustration by Dominic Bugatto

It might be comforting to assume that, a century later, historians, the general public, and even the impressionable teenagers of the early 22nd century will still have access to all the videogames created to date. In this digital age, it's tempting to believe copies of the videogames you've helped create are never going to disappear. But it's not completely clear that'll be the case, especially with titles from the 1980s and early 1990s preserved on decaying magnetic media without a central repository to officially archive game titles and other software products. So is there anything we can do? Are copies of some vintage games in danger of just disappearing forever?

The End is Near

Since the average lifespan of magnetic media such as floppy discs has been estimated at 10 to 30 years, one thing is clear: time is running out. You might ask, hasn't all the archiving already been done? Aren't the legally dubious abandonware web sites such as The Underdogs safeguarding the industry's history? Although they don't actually have the rights, aren't they keeping digital copies of a lot of classic games developers and publishers have forgotten about completely? Well, yes, in many cases they are, but the uneasy standoff between those trying to enforce copyright laws and those who are happy (or oblivious) enough to let their old titles live on unofficially makes it impossible to have an authoritative archive made up of posted-on-the-web disk images. You can't create a reliable, permanent archive of game-related materials that are publicly distributable, because copyright crackdowns will (quite rightly) make such venues go away. In some cases, people may use abandonware sites as a sneaky way to justify piracy.

Well, here's another option. Could we expect game developers, publishers, and right-holders to do all the work? Realistically, with economic constraints and the massive amount of complex company consolidation that goes on, most publishers do not even have a comprehensive list of videogames they own the rights to. Even if they do, it's simply not worth their while (in their view) to archive the vast majority of the titles. With the increasing popularity of classic-retro compilations, publishers are certainly revamping some old titles for new consoles, but they're often changing and adapting along the way. Many good but obscure titles don't receive such treatments. You can't expect companies to archive their entire catalog for no monetary gain, even if they have the will to do so. Elsewhere in the technology industry, the most forward-thinking and history-conscious companies, such as IBM, keep official archivists. Still, there are some things—such as original production materials, source code, and other internal artifacts—that just can't be preserved by outside institutions, because they don't have access to the data.

Institutional Approaches

The ideal would be to have a massive institutional collection of games and other retail software, with all the original physical artifacts (box, manual, discs, and so on) stored in a safe place, and a digital database with exact copies of the data on the disc (not just illegal "cracked" images), stored for safekeeping. For each game in the collection, you'd have a perfect digital copy of it, so when the floppy discs are no longer readable, there's a private copy of the data stored for posterity. In addition, the data can become available for public distribution whenever copyright (finally) runs out and the program becomes public domain. In some cases, the right-holder may decide to permit copying.

Some institutions are starting to make a move toward preserving software properly. Stanford University has started a game-preservation initiative. The Stephen F. Cabrinety Collection in the History of Micro-Computing includes as yet unpreserved retail software, largely videogames. Amassing well into tens of thousands of classic games, it's probably the largest

continued on page 71

continued from page 72

institutional collection in the world. The Computer History Museum in Silicon Valley is starting to look at software preservation seriously too. But it's going after largely software languages and packages (some of which are also in imminent danger of being destroyed) unrelated to games.

Legal Issues

I've been working with the Internet Archive (www.archive.org), a nonprofit institution that's "building a digital library of Internet sites and other cultural artifacts in digital form." We discovered possible archiving issues involving the Digital Millennium Copyright Act (DMCA), which may have made it impossible to legally archive early computer software and games, even for accredited institutions wishing to store limited amounts of private, non-circulating, archival images. So we petitioned the Copyright Office about these access protection issues, and the U.S. Copyright Office ruled in October 2003 that exemptions should be added to the anti-circumvention clause of the DMCA, to be valid until the next Copyright Office rulemaking in 2006 (www.copyright.gov/1201/docs/librarian_statement_01.html). The exception applies to "computer programs and videogames distributed in formats that have become obsolete and which require the original media or hardware as a condition of access."

This does not mean titles posted as abandonware are legal to copy as you please, but it does arguably mean official institutions can make a limited amount of private archival copies of classic software, provided they own the original physical copy, and their copy doesn't violate the DMCA. So the possibility now exists for good archiving to happen, and we're in the early stages of starting software archiving projects. Bear in mind, the Internet Archive does not claim to be the sole solution—just one of many possible contributors, especially now that the DMCA exemption has arguably made classic game archiving legal in the U.S.

This situation needs a critical mass of developers like you bringing your technical knowledge to bear on the complex archival problems. Your efforts may include donating old retail software for archiving or even allowing some of the less financially important titles in your company's back catalog to become freely available through the archives. If there's playable public content in these putative software archives, alongside good metadata and information on the private content, then the pieces will be in place to create a canonical archive of games, ensuring the titles you worked on won't disappear.

SIMON CARLESS | *Simon is a former videogame designer (Eidos, Atari), who now edits the popular tech web site Slashdot (www.slashdot.org). He can be contacted at scarless@gdmag.com.*

ADVERTISER INDEX

COMPANY NAME	PAGE	COMPANY NAME	PAGE
Academy of Art College	63	Iron Lore Entertainment	59
Academy of G.E.T.	62	Law Offices of Michael H. Golland	70
Activision	54	LucasArts	52
Aladdin Knowledge	5	Metrowerks	14
Alias	11	Monolith	52
Alienware	3	New Pencil	69
Araxis	32	NXN Software	39
Art Institute of Vancouver	42	Omni Interactive	70
Atari	55	Perforce Software	13
ATI Technologies	35, 17A	Premier Press	25
Audio Images (Wacom)	19	Programmer's Paradise	9
Carnegie Mellon University	67	RAD Game Tools	C4
Castles Music	70	Rainbow Studios	52
CCP Games	68	Rtzen Inc.	C2
Charles River Media	41	Sammy Studios	58
Climas Group	57	Savannah College of Art & Design	61
Collins College	66	Softimage	C3
Cranky Pants Games	52	Sony Computer	56
Cyberware	21	The Art Institute International	64
Fairfax Games	60	The Collective Inc.	54
Full Sail Real World Education	62	The Hartecenter at SMU Guildhall	61
IDA of Singapore	7	Trymedia Systems	31
Inevitable Entertainment	54	University of Advancing Technology	65
Integrityware	68	Vancouver Film School	62
Intel	53	Vicon Motion Systems	27