GAME DEVELOPER MAGAZINE

FEBRUARY 1998

Heart and Soul

"People ask me whom I fear… which of our competition — LucasArts, Microsoft, any of the big companies. They don't frighten me. What I'm afraid of is two guys in a garage, working in total obscurity. That's where the heart and soul of this business is at." – Jay Wilbur, during his days at id Software

This quote, which appeared in *Wired* magazine a couple of years ago, was a shot in the arm for many small game developers hoping to make it big. And if it wasn't apparent from the millions of newbie questions that pepper rec.games.programmer, there are many such people.

A common question posed by newbies to insiders is, How do I break into the game development industry? If you've ever tried to answer this one, you've probably come to the same conclusion as me and realized that there's no one particular path into the industry. It seems everyone entered through one back door or another. Only recently have a handful of schools, such as Vancouver-based DigiPen (www.digipen.com), begun to offer degrees in game programming. But for most people, landing a good game development job is quite a game in itself.

So naturally, I was happy to get a call from a group of Berkeley undergrads who are taking a bootstrap approach to their career development. These students formed a game development co-op in 1995 called the Freelance Gaming Studio (FGS) with the intent of developing and distributing a multiplayer Internet-based game. This team wants to take their game — simply titled THE FISH GAME (www.fishgame.net) — out of the lab and into the real world. The game, which FGS demo'd at the CGDC last year, is a bit like Virgin's SUBSPACE, but set in a tidepool. Your fish (either a ray or a puffer fish) is viewed from an overhead perspective, and you use the keyboard to steer it around in the water, avoiding or attacking other aquatic life as you desire. The game will include scenario editing tools and possibly server capabilities so that anyone can host games over the Net.

There are approximately 40 FGS developers, and a core group of 15 active members meets on a regular basis during the school year to assign development duties, check on the progress of various features under construction, and of course to engage in a little "playtesting." FGS doesn't plan on charging for the game when they complete it — they intend to distribute the game freely. The goal is to build interest in the game via the Net, build player interest and loyalty, and parlay the popularity into a possible business venture with a publisher. After meeting with FGS president Jason Chein though, I get the impression that he'll be happy regardless of the future commercial prospects of THE FISH GAME, because of the experience that he and the team gained in developing it. They're not blind to the fact that developing the game substantially improves their prospects when they start hunting for jobs, and may even let them continue to work together after graduation as an independent, professional development studio.

I can't help but admire what the FGS is attempting. Whether or not their game is successful is really beside the point. As students, they have the luxury of ignoring quarterly earnings reports, shareholders, and even payroll. They're in it for the fun and the experience, and any kind of commercial reward is really just a bonus. Chein wants to take his model of the student-run game development studio to other campuses so that students elsewhere can get the same game development team experience.

On a final, somewhat eerie note, THE FISH GAME proved to be an educational opportunity for some business administration students at Berkeley as well. A team from the BA161 class used THE FISH GAME as a means of studying product marketing and distribution. Another example which illustrates that *no* game can escape the grasp of the marketing department, eh? ■

Alex Dunne

EDITOR IN CHIEF Alex Dunne
adunne@compuserve.com
MANAGING EDITOR Tor Berg
tdberg@sirius.com
EDITORIAL ASSISTANT Wesley Hall
whall@mfi.com
EDITOR-AT-LARGE Chris Hecker
checker@bix.com
CONTRIBUTING EDITORS Brian Hook
bwh@wksoftware.com
Josh White
josh@vectorg.com
ADVISORY BOARD Hal Barwood
Noah Falstein
Susan Lee-Merrow
Mark Miller

COVER IMAGE Bioware

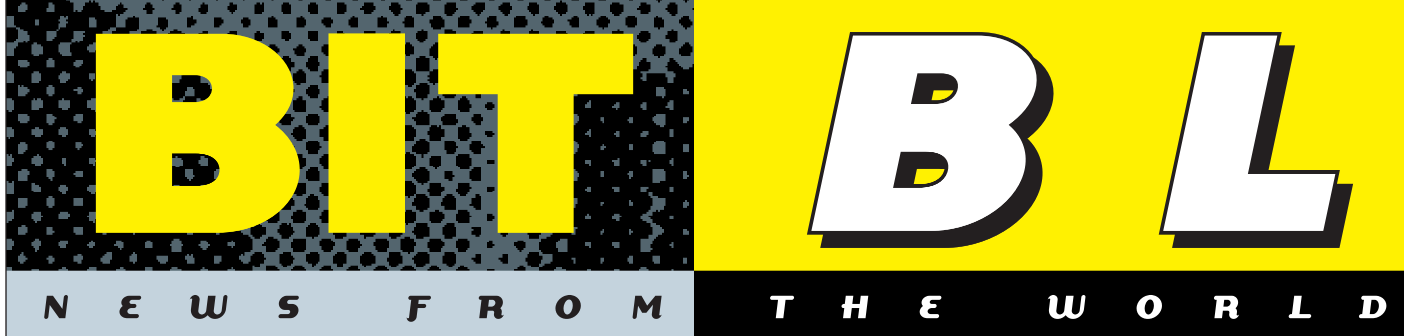
PUBLISHER KoAnn Vikören
ASSOCIATE PUBLISHER Cynthia A. Blair
(415) 905-2210
cblair@mfi.com

MARKETING MANAGER Susan McDonald
AD. PRODUCTION COORDINATOR Dave Perrotti
DIRECTOR OF PRODUCTION Andrew A. Mickus
VICE PRESIDENT/CIRCULATION Jerry M. Okabe
GROUP CIRCULATION DIRECTOR Mike Poplardo
CIRCULATION MANAGER Stephanie Blake
CIRCULATION ASSISTANT Kausha Jackson-Crain
DIRECT MAIL MANAGER Claudia Curcio
NEWSSTAND MANAGER Eric Alekman
REPRINTS Stella Valdez
(916) 983-6971

Miller Freeman
A United News & Media publication

CEO - MILLER FREEMAN GLOBAL Tony Tillin
CHAIRMAN - MILLER FREEMAN INC. Marshall W. Freeman
PRESIDENT/COO Donald A. Pazour
SENIOR VICE PRESIDENT/CFO Warren "Andy" Ambrose
SENIOR VICE PRESIDENTS H. Ted Bahr,
Darrell Denny,
David Nussbaum,
Galen A. Poss,
Wini D. Ragus,
Regina Starr Ridley
VICE PRESIDENT/PRODUCTION Andrew A. Mickus
VICE PRESIDENT/CIRCULATION Jerry M. Okabe
SENIOR VICE PRESIDENT/ SYSTEMS AND SOFTWARE DIVISION Regina Starr Ridley

www.gdmag.com



INDUSTRY WATCH

by Alex Dunne

THINGS THAT MAKE YOU GO HMMM. To those who doubted that Microsoft would ever take steps to further support OpenGL development under Windows, the joint announcement between Microsoft and SGI about their new OpenGL device driver kit seems to have been a welcome shock. The DDK, to be distributed this spring by Microsoft, will include a new OpenGL ICD developed jointly by the two companies. There will also be a certification and logo branding program for both OpenGL and Direct3D drivers created with the DDK. The press release took the Microsoft party line in its repeated references to the "professional 3D" uses of OpenGL and to the "consumer application" uses of Direct3D. Despite the specifics of the wording, however, it seems to be a win for OpenGL developers, many of whom will continue to keep a vigilant eye on Microsoft to see whether it drags its feet in development and certification of OpenGL drivers.

CARMACK CONSIDERING JAVA. Those who follow Carmack's .plan may have caught his recent comments about Java "playing a significant role in future projects" for id. The fact that id ports its titles to so many different platforms must make the language an attractive alternative with which to create game utilities. Carmack stated that the number crunching utilities were likely to remain in C, however.

HEAVEN HELP US. Mike Wilson's departure from the helm of Ion Storm to form a company called G.O.D. (Gathering of Developers) got many chuckles around our office, and presents Wilson with the

Alias Renderer for Windows NT

ALIAS|WAVEFRONT is now shipping the Alias Renderer for Windows NT. This is the same film-quality renderer originally developed on IRIX — now ported to the Windows NT operating system.

The Alias Renderer for Windows NT will read SDL files created by PowerAnimator, Studio, and Designer. If you're familiar with the command line renderer on IRIX, you already know the interface. The renderer is multi-threaded and will be available in different versions that support one, two, four, or eight processors in a given NT workstation. Both node-locked and "floating" versions of the renderer are available. Output (rendered image) files are SGI compatible, plus the product includes the fcheck image viewing utility for use on Windows NT. Alias|Wavefront's tests indicate that a 200 Mhz Pentium Pro based system will be slightly faster for rendering than an O2 R5000 running at 180 Mhz. Dual Pentium IIs running at 266 Mhz (Powercaster) will generally outperform an R10000 Octane running at 195 Mhz.

Jellyfish image created with Alias Renderer for Windows NT.

To run the Alias Renderer for Windows NT, you'll need a Pentium-based workstation (133 Mhz or higher), Windows NT 4.0, 128MB of RAM, 100MB of free disk space for installations and space for renderings, and a CD-ROM drive. Pricing starts at $1,995. For more information, call your local sales representative as listed on the Alias|Wavefront web site.

■ Alias|Wavefront
Toronto, Canada
(416) 362-9181
www.aw.sgi.com/pages/home/index.html

DeBabelizer Pro 4.5

EQUILIBRIUM has unveiled the latest upgrade of their automated graphics processing software, DeBabelizer Pro 4.5 for Windows 95/NT.

The premise behind DeBabelizer's automated graphics processing technology is that designers need to efficiently and automatically process content so that they can deliver graphics across a range of platforms while maintaining the highest level of image quality. DeBabelizer Pro automatically prepares images, animations, and digital video through intuitive drag-and-drop scripting, batch processing, color palette reduction, image processing, and file-format conversion. New features include the ability to run multiple batch processes consecutively; to outline or shave a one-pixel perimeter around an object on a solid background; to composite a batch of images or frames against another batch; to compare batches to identify pixel differences; to apply any image process to multiple frames; to sort and render

batches; to track all processes while running unattended batches with the new global log; and to increase processing throughput by 40 percent when display mode is off. In addition to over 100 currently supported file formats, DeBabelizer Pro 4.5 now supports .PNG, .EPSF, and reads and writes Kodak's FlashPix format.

DeBabelizer Pro 4.5 supports Windows 95/NT and has a suggested retail price of $595.

■ Equilibrium
Sausalito, California
(415) 332-4343
www.equilibrium.com

GameMix

MEDIA SUPERCOLLIDER, creator of Java-based multiplayer games, has announced GameMix, its first release of multiplayer game server software.

GameMix provides Java programmers with an easy-to-use toolkit for creation of high-speed, multiplayer games. GameMix allows developers to focus on the game itself, and won't double the development period on network implementation. The server can accommodate more than 64 simultaneous users and is very light on system overhead. Media-SuperCollider has written two games

that utilize GameMix server technology: BATTLETANK (3D) and FIGHTINGVERGE (2D). Written in Java, the games may be played on virtually any platform.

GameMix may be downloaded from www.gamemix.com for a free 30-day trial. In addition, developers are invited to utilize Media SuperCollider's servers as a development/game hosting site for GameMix games. With this service, smaller game companies will have access to powerful Sun servers and a high-speed connection to the Internet. It sells for $299.

■ Media SuperCollider
Marina del Rey, California
(310) 448-4171
www.mediasc.com

Power Render 2.5

EGERTER SOFTWARE has just released Power Render 2.5, the next major version of their game development libraries.

This release adds Direct3D support for 3D cards, in addition to the already existing support for VGA, SVGA, 3Dfx's Glide, and DirectDraw. A graphical user interface library allows full-screen software-rendered or 3D-accelerated utilities to manipulate 3D objects and view scenes as they would appear in the final application. Power Render allows for rapid importing of objects, animations, camera paths, and scenes from 3D Studio and Lightwave. The utilities let the designer apply textures, material properties, and rendering methods without recompiling any code. You can actually build and view a fully textured and animated scene in real time without touching a compiler. Source code for the utilities is provided so developers can modify them or use them as a basis for new utilities.

Power Render 2.5 has a suggested retail price of $299. The libraries and utilities are available for free trial use from the Power Render web page.

■ Egerter Software
London, Ontario, Canada
(519) 641-7542
www.egerter.com/pr

opportunity to assume the title of "Right Hand." Be wary how much you jest about this company, though, lest you incur its wrath.

MORE UO NEWS. At the Online Games Conference in Los Angeles last November, one of the best talks of the event came from Richard Garriott. Garriott addressed some of the criticisms about the game and admitted that it was still somewhat of a work in progress. Demand for the game, which has been EA's most rapidly selling PC game in the company's 15-year history, far outstripped what Garriott and other UO developers anticipated. Sun servers have been added at a furious pace to keep up with the load (a problem many companies would kill to have). Separately, EA also announced Game Time, a 12-page brochure for $29.85 that contains a registration code and player tips. With it, players can buy 90 days of prepaid online access to the game with cash or check at Babbages, Electronics Boutique, and Egghead. This smart move will make the game more accessible for younger players who don't yet enjoy the privilege of getting a monthly Visa statement.

PORT IT (WELL) AND THEY WILL COME. Eidos acquired from Square the exclusive North American and European rights to publish FINAL FANTASY VII for the PC. FINAL FANTASY VII for the PlayStation, with more than 3.2 million units sold in Japan, is the best-selling PlayStation title. Assuming that the port does justice to the game (gripes about the RESIDENT EVIL port come to mind), landing this already heavily marketed title is a big win for Eidos.

CHAPTER 11 CITY. GameTek, the Sausalito, California-based developer and publisher of DARK COLONY for the PC and WHEEL OF FORTUNE and JEOPARDY for the N64, filed a petition for reorganization under Chapter 11 bankruptcy. The company blamed heavy losses, development delays and disappointing sales.

So Long and Thanks for the Rail Gun

As this will be my last Graphic Content, I wanted to wrap up my discussion of my experiences working on Quake 2 with an examination of the tools, both software and hardware, that we employ at id. I'll finish with a diatribe on glad-handing and wasting time.

Development Software

While I'm going to discuss the software development tools used at id, this is not necessarily an endorsement, or even very useful information. Still, it may satisfy the curiosity of many up-and-coming developers. id software has used a vast array of programming and development tools throughout its history. WOLFENSTEIN 3D was written using Borland C++ 3.1 (16-bit real mode DOS), DOOM was written using Watcom C/C++ 9.x and DOS4GW (32-bit protected mode DOS), and QUAKE was written using *gcc* and the CWSDPMI extender (32-bit protected mode DOS). id's first official Win32 products, WINQUAKE and GLQUAKE, were written using Microsoft Visual C++ 4.x.

QUAKE 2 was developed using Microsoft Visual C++ 4.2 and 5.0 for Win32. On Linux and Apple Rhapsody, we used *gcc,* while on Silicon Graphics IRIX, we used the standard MIPS *cc* compiler. For Win32, we chose Microsoft Visual C++ because it generates pretty efficient code, is well supported, is written by the same company that creates the operating system we're using and targeting, has a nice IDE and help system, and isn't going away anytime soon. Oddly enough, the entire programming staff uses the Microsoft Visual C++ IDE for editing chores instead of a third-party editor and *make* utility.

For debugging, we've relied primarily on Microsoft Visual C++'s internal debugger, but we've had our asses saved in several situations by using NuMega BoundsChecker. I cannot recommend this product highly enough — it has saved us dozens of hours and probably knocked at least a week off of our development time. The value derived by avoiding the demoralizing effect that a show-stopper bug can have is immeasurable. It's a simple fact: BoundsChecker is easily a no-brainer development tool if you are developing for Win32.

For profiling we've mostly relied on our own internal profiling code, but towards the end of the project we started using Tracepoint's HiProf for C++ product, which has turned out to be really good. We've also occasionally used VTune from Intel. We never managed to get Microsoft Visual C++'s profiler to work for us. For now, we're quite happy with HiProf for C++.

Our primary development platform is Windows NT 4.0 (Service Pack 3), and we're quite happy with it. It's both robust and reasonably compatible with Windows 95. The vast majority of the crashes that we've suffered under Windows NT have been due to driver bugs in manufacturer's GDI and OpenGL drivers (which run in kernel mode and can thus trash the system). There is a certain sense of calm that you get when writing code on an operating system that you know isn't going to barf simply because you moved the mouse while the OS is going through the software equivalent of a mid-life crisis.

However, both the programmers and level designers have at least one Windows 95 machine, typically a reasonably slow one, that is used for testing, debugging, and performance measurement.

As for external system libraries, we use OpenGL and DirectX —but neither

Well, this is the last of my regular columns. I've had a good time writing these, but since taking on the task of being a columnist for Game Developer, a "real" job has reared its head. I plan on writing more articles in the future, as time permits. I also encourage other game developers to write down their experiences and share the wealth of knowledge they've accrued over the years.

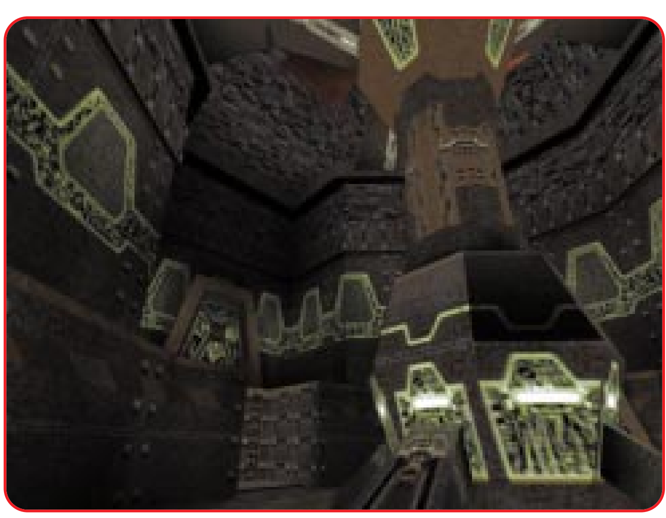
of these components is required. We use `LoadLibrary()` to check for the availability of DirectDraw, DirectSound, and OpenGL. If DirectDraw isn't available, we use GDI DIB sections for software rendering. And if DirectSound is unavailable, it is substituted with WAV out. If no OpenGL driver is installed on the target system, then we resort to software rendering.

Development Hardware

All of our front-line developers (level designers, artists, and programmers) are equipped with Intergraph TDZ-410 Windows NT workstations with Intergraph Realizm graphics adapters. These are dual processor Pentium Pro 200 machines, each with 4GB hard drive, 128MB of RAM, and Nokia 445Xi 21-inch monitors. We really like the Intergraph systems, even though they are considerably more expensive than comparable clones, because they are sturdy, compatible with existing software, and very fast. The Intergraph Realizm drivers are easily the most robust and well implemented OpenGL drivers that we've used, and the built-in peripherals (sound, mouse, keyboard, SCSI, and Ethernet) just work. Not enough can be said for a machine that actually works all of the time.

John Carmack has a slight variant on the workstation just described. He has the same basic Intergraph workstation, however, he was two Intergraph Realizm graphics adapters driving dual 21-inch flat panel LCD monitors instead of a single Realizm driving a single CRT. This setup definitely has its advantages, including the ability to run your application on one monitor and your debugger on the other. Sometime after Christmas, we'll probably upgrade our primary development systems to dual processor 300Mhz Pentium 2 class machines, 24 inch monitors, 256MB of RAM, 27GB hard drives, and graphics subsystems with (hopefully) at least 50% more fill rate than the existing Realizms.

Kevin Cloud and Paul Steed, our resident artists responsible for modeling and animation, use SGI workstations so that they can run Alias|Wavefront. One of the workstations is an SGI O2 (MIPS R5000-based), and the other is a somewhat long-in-the-tooth SGI Indigo2 Extreme. We had originally intended on transitioning to a Windows NT-based modeling package such as Softimage or 3D Studio MAX, but the artists love Alias so much that we're going to be stuck using SGI machines for a while.



Our monster CPU server is an SGI Origin2000 machine with sixteen MIPS R10000 processors. This machine runs IRIX and is responsible for executing all of John Carmack's custom tools, including our BSP generator, visibility calculator, and radiosity lighting tool. All of the QUAKE 2 tools are built with parallel processing in mind, and have been since id used a four-processor DEC Alpha machine for QUAKE development. The SGI Origin2000 has replaced the quad-Alpha machine used for QUAKE's development, and is roughly six times faster overall. Our Win32-based level editor has a menu option allowing the level designers to *rsh* commands to the Origin2000 from their Intergraph workstations — they can then immediately continue working on their levels while waiting for their jobs to finish on the compute server. We expect the Origin2000 to have a useful life through Trinity, but after that we'll probably switch to something else, hopefully something that's based on DEC Alpha processors and Windows NT. While running many different operating systems is neat from a "gadget" perspective, it really plays hell with productivity. Having at most one or two different operating systems in your critical path makes for fewer headaches when trying to get real work done.

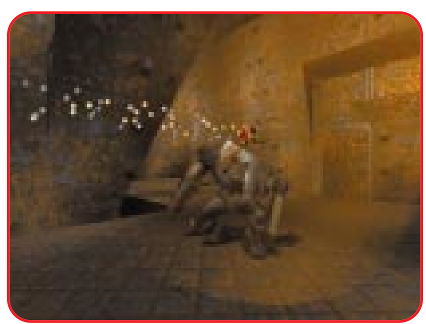
A single shared file server, consisting of a large (~70GB) DEC disk array accessed through a regular Intel-based Windows NT Server machine, is used for globally accessed data and executables. Through most of QUAKE 2's development we were networked on 10Mbit, but at the very end we switched over to

100Mbit Ethernet, which has noticeably improved productivity, since we manipulate extremely large data sets. During the development of Trinity, we'll be switching to 100BaseT as our office standard, which opens up a lot of possibilities (including storing our home directories remotely over the network).

Quality Assurance and Testing

id software doesn't have a comprehensive suite of test machines. We're actually borderline irresponsible when it comes to testing — we develop on our nonconsumer Windows NT workstations, and hope that if we "code well," that things will just magically work when we run on the many incarnations of Windows 95, even with screwed-up drivers. The scary thing is that this works most of the time.

So far, we've found only a couple of irritating differences among Windows NT, Windows 95, and Windows 95 OSR2.x, and those types of incompatibilities are quashed within a day or two. In the future, we're going to try to make things more robust by having some strategy in place when configuring sec-



ondary performance/compatibility systems for programmers and level designers. Right now we rely heavily on our publisher to find the more obvious kinks in our code, but in the future, we hope to manage the QA and testing procedure a little more responsibly.

Developing Relationships with Hardware Vendors

This may seem like an odd topic to talk about, but I'd like to explain how we manage our relationships with hardware vendors, specifically graphics card and chipset manufacturers. There is a wide misconception that id software receives special treatment from hardware vendors. Yes, in some cases they approach us and want us to help them (note that this is not the same as them wanting to help us), but often as not we have to beg and plead for development boards, driver updates, bug fixes, and workarounds. id has a lot of visibility within the industry, but in terms of pure dollar sales, we are hard-pressed to compete with larger (publicly owned) companies.

I regularly spend a few hours a day sending e-mail, answering e-mail, testing graphics adapters, and making phone calls to hardware vendors, all in the name of bug fixes and coming up with workarounds. I've devised some important strategies and tactics from working with many different hardware manufacturers. For starters, never tell them that a bundle deal is out of the question. Even if it *is* out of the question, you're not doing yourself any favors by turning off product managers and OEM sales managers at the outset. These types are constantly hunting for good quality titles at a good price to bundle with their hardware, and if you can keep them interested long enough to make sure that whatever bugs you need fixed are taken care of, then you're doing pretty well. My favorite tactic is to say, "We won't be able to discuss a bundle deal until we see how well your hardware performs with our title, and in order to do that we need just these few bugs fixed…." Works like a charm (mostly because it's true).

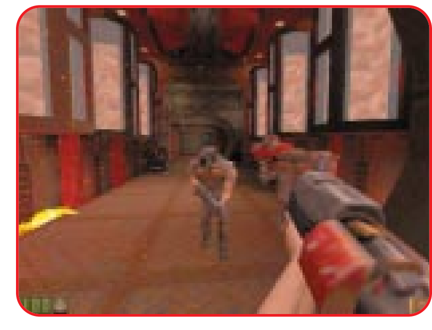
Speaking of sales and marketing types, believe it or not these guys can often be your allies. As a general rule, until you start getting positive responses from the engineers at a company, keep everyone you know at that company in the loop — sales, marketing, developer relations, engineers, interns, just about anyone with an e-mail address you can dig up. If you send e-mail to five people but only one is interested in helping you, then that one individual can often make the difference when it comes to getting a response or not — especially if that individual is important. Marketing, sales, public relations, and

similar droids-in-suits are often very willing to smack reticent support personnel on your behalf. The key is that you need to make sure that you're talking to the right people from the get-go. Nothing is more frustrating than only having a single point of contact who couldn't care less about you and your product that "doesn't ring a bell."

Fear is also your friend — fear of making someone look bad. The carrot and the stick approach works, especially if you wield even a modicum of power. If asking nicely and being persistent doesn't generate results, then don't be afraid of letting it be known that someone's hardware is "unsuitable" for your title via a Usenet posting, a .plan file update, a web page comment, or even an interview with a magazine. This will often get results *very* quickly. I've only done this on a couple of occasions, and it's worked really well the few times I've resorted to it. The downside to this tactic is that you run the risk of angering the people from whom you're trying to get assistance, so only use this as a last resort. Always try to be diplomatic if at all possible. For example, "We've found some problems with XYZ's drivers, and we're not sure if we'll be able to find workarounds in time for our product's release," is much better than what you might *really* want to say — "XYZ's driver is completely broken, and the cubicle monkeys there are too stupid, lazy, and arrogant to even return our e-mails, so screw 'em."

"Coolness" is one of those intangible assets that many game developers have, mostly in the form of a popular title, and this can be leveraged into good relationships with driver engineers. I've actually been on the other side of this coin. I practically begged Parallax to let me port DESCENT 2 to the Voodoo Graphics accelerator when I worked at 3Dfx. These kind of people exist at just about every hardware company, and if you can find them you've struck gold.

If you expect hardware developers to take you seriously though, you *must* help them out as much as possible. I've heard all too often of cases where ABC Software claims there's a bug with XYZ's driver, and they'll rant about how crappy the driver engineers at XYZ are, but when pressed to send in a code snippet or test program, they'll balk with some excuse such as, "We don't have time" or "We can't send you our .EXE, we don't want it leaked." If you're not willing to expend some effort, then don't be surprised if the hardware company isn't either. Many times, software developers expect hardware vendors to find driver bugs based on nothing but a general description of what happens: "I get weird red streaks whenever I switch to this video mode."



Uh, yeah, great, thanks for narrowing it down. You'd be surprised at how responsive the much-maligned driver engineer can be if provided with a short chunk of code that reproduces a bug in their code.

I would say that the single most important trait to have when working with hardware developers is persistence. You *must* keep sending bug reports, asking for status updates, making phone calls, and doing whatever else you can to make sure your problem is being handled. If you suddenly shut up, many people will assume that you've either lost interest or found some workaround.

Productivity

One of the most important things that I've learned in the past few years is that the number of hours you work does *not* necessarily correlate to your productivity. Productivity is based on the work that you accomplish, not the hours that you work. While the two can be related, in surprisingly many cases, they are not. Analogy: someone who writes code for only one hour a week is still more productive than someone who surfs the Web, plays DIABLO, makes personal phone calls, reads e-mail, and takes three-hour lunch and dinner breaks for sixty hours a week.

One source of time suckage that id doesn't force on employees is long-winded company meetings. As a matter of fact, in the four or so months that I've been here, we have yet to have a single company-wide meeting. Most meetings that take place consist of informal conversations between two or three guys in a hallway or the break room. These types of meetings can be work related or just chewing the fat, but inevitably useful information is disseminated. A "real" meeting usually only has to occur when some amount of interactivity is necessary, and in our case only rarely does the entire company need to get together to address some issue.

John Cash, John Carmack, and I probably talk for a couple of hours each day, and we often go to dinner together, where we have the chance to both relax and talk shop without taking time away from coding. This keeps us in sync technically, and if any of us have gleaned some important information from the artists or level designers during our daily routine, we propagate it amongst ourselves — very simple, very efficient, and we never have to set an alarm clock.

On occasion, Carmack and I have to meet with external parties such as hardware companies and occasionally press types. However, we try to minimize the frequency of these types of meetings. When a meeting is unavoidable, we adhere to a very simple "one-hour rule" — the meeting cannot run longer than one hour. This prevents meetings from degenerating into a "let's get to know each other and have warm fuzzy feelings" productivity vortex, and it works extremely well at keeping the visitors focused on their agenda.

A meeting can often be just as easily handled with a conference call, and a conference call can usually just as easily be handled by a short e-mail exchange. In some cases, such as when a high bandwidth conversation needs to take place, a phone call or meeting works much better than e-mail. But in most cases, e-mail provides all of the bandwidth necessary to communicate ideas effectively.

One thing that Cash, Carmack, and I do is post publicly available work logs (via the infamous .plan file). This lets us sit back at the end of the day and say "I got a lot of work done," or "I screwed off for an entire day." Knowing that you are accountable to all your coworkers, consumers, and fans makes you far less likely to chit-chat on the phone for hours on end.

Conclusion

Hopefully, this article has illustrated the fact that nothing magical happens at id software that gives id an unfair edge over the rest of the industry. The formula for success applies the same to id as it does to everyone else — talent, hard work, and keeping focused are all that matter. Shrewd business skills also help, although even that isn't always necessary. If you have the skill, the ambition, the drive, the focus, the energy, and the time to do a good game, you should easily outclass your competition. id got where it is today without making a pact with the devil (at least, no one's told me so far if that's the case), and if id releases a product that stinks, I'm quite certain the market will let us know. ■

Wanted: APIs, Hold the Code.

After several sessions of frustrating Web surfing, I came to a realization: game developers learn how to use new core technology in lots of ways, but for non-coders (artists, designers, producers, and so on) one learning path is noticeably absent — the horse's mouth.

Core technology developers don't seem to know that artists are capable of understanding — and care about — their offerings. They address programmers almost exclusively, and I suppose that they expect the content development methods to trickle down. Or maybe they think it's too hard to explain without using code. I wish that the companies developing new core technologies would create information sources, such as Application Programming Interface (API) documents, for artists and designers as well as for programmers.

When I say "core technology," I'm talking about fundamental improvements in game development, such as advances in 3D graphics hardware or new APIs for force feedback joysticks. Technologies that I'd particularly like to see improved are DirectX, OpenGL, and most game-centric 3D graphics chips.

Besides teaching artists who are already using their technology, an API-type document has another important function: it's in technology developers' best interests to give artists and designers the chance to explore these new ideas firsthand. Noncoders are often the ones who see creative new uses for new ideas — they're the usually the ones who hold the creative vision that needs to be expressed, and if they really understand emerging technologies, they'll be in a position to make killer applications that rely on the core technology.

These documents should be written for noncoders, ideally by other experienced, articulate noncoders (or at least really good tech writers who actually use the product). Of course, the authors need convenient access to the key technology developers, as well as plenty of support (such as illustrators, reasonable time budgets, and so on).

There are no standard paradigms, let alone commercial tools, that generate truly interactive animation. Our tools represent animation as sequences of preset poses, like a movie.

What would be in these documents? No marketing hype — instead, the documents would contain exact procedures for building content using the technology, in casual, clear language. I'm hoping for a simple, understandable document structure with before/after image examples, and imaginative demonstrations of what the technology will allow. Excellent distribution would also be great. I started a template outline of what I'd like.

If you're interested in this idea (or if you've seen such documents that already exist), e-mail me at column@vectorg.com.

Low-polygon Character Animation

Writing for magazines is so weird. Not only am I writing this article three months in advance, but we've been building this character for months now. In this industry, that's long enough for the whole paradigm to change. It's like painting the Golden Gate bridge: by the time we get to the end, we should redo the beginning. Of course, the basics don't change very fast, so I won't get depressed about the fleeting value of written knowledge if you don't.

First we'll look at interactive animation, examine some specific terms and issues that are unique, and deal with building animations that loop and branch well. Then we'll discuss Jim, our animation example and wrap up.

As 3D animators, you should already know about the basics of 3D animation (if not, I recommend doing your modeling software's animation tutorials), so I'm going to focus on the harder, murkier stuff that happens when we build animations for interactive games.

Why does this topic need its own article? Real-time 3D (RT3D) animation is hauntingly similar to normal prerendered animation — and yet, it has a different purpose, we face different issues, and we struggle with different use of our tools.

Interactive Animation

"Interactive animation" is the kind of difficult issue that the game designer/lead programmer has usually worked out prior to production — however, the astute professional artist (that's you) definitely should be

Terms You Ought to Know, Part 1

As I've cheerfully prattled about "transforms per frame," I've been wondering: does my beloved audience have any clue what I mean by "transform"? Are they thinking of a different meaning for "frame" than the one I'm referring to? Have I strayed too far into the fog of technical vocabulary?

But whew! This section is letting me rest easy.Here, I'll explain exactly what these terms mean to me. Since there's no way to define these terms abstractly, I'll spin some fascinating examples of their misuse. And, as a special bonus to all you diligent sidebar readers, I'll work a little sex and violence into the examples.

TRANSFORM. A transform is a set of numbers that can define a position (x,y,z), rotation (roll, pitch, yaw), and scale for a single object. You probably knew position no problem, but scale is a weird one. It includes squash/stretch as well as uniform size change.

As a verb, it's often said in a rather uptight phrasing: "Obj01 applies a transform to Obj02" I prefer this to the easy-but-ambiguous "Obj01 transforms Obj02," because transform has too many other meanings in English; with the first phrasing, there is no doubt that we're talking about a numeric transform.

It takes nine real numbers to store a transform. Considering the number of transforms needed to animate, that can really add up. Being performance-oriented, we immediately notice that we can drop some of these unnecessary numbers. This reduction is called a "constraint" because if we get rid of a number, that means we can't animate what it stored. For example, if we get rid of the three scale numbers, our object can't grow or shrink during the animation: we are *constrained* to using a single size for our object.

CONSTRAINT. You've probably used constraints in prerendered 3D modeling software. They work similarly, but have a different purpose. During normal prerendered animations, constraints are useful as a convenience for limiting motion to a reasonable range. For example, we'd like to prevent a knee joint from bending backward or moving away from the body, so we constrain the joint to rotate only part-way around one axis, and eliminate the other rotation axes as well as all position and scale.

Most of our animated objects are constrained in some way. In fact, most character animation uses only joint rotations for their transforms. Motion capture data in particular is usually stored as joint rotation only. For example, a knee only requires a single rotation value in its transform.

This is a good point for a quick reminder that each joint is relative to its parent: if we constrain a knee to a single bend, but then we zoom the whole body through space like Superman, the knee will stay connected, zooming through space as it should.

Obviously, constraints are good for storage, but it means that we can only show the knee in natural motion — if the player wanted to rip his lower leg off and hurl it at an opponent, the programmers would probably have to unconstrain it so it could have full rotation and movement, which might mean storing more data in the transform.

---

familiar with the problem.

Why is "interactive animation" such a nasty issue? Because animation isn't really interactive. OK, that's not quite true: theoretically, animation can be truly interactive, but it's a job for a really, really good programmer. Few artists are technical enough to design animation in "pure" interactive format because it is written in C++ — there are no standard paradigms, let alone commercial tools, that generate truly interactive animation.

Our tools represent animation as sequences of preset poses, like a movie. Animation is much weirder in real-time 3D. We're constantly struggling to find the balance between giving the users unscripted freedom (letting them control events) and artistic vision (playing our gorgeous sequence of scripted animations). Here are a few of the various ways to integrate these two ideas:

NO INPUT (SCRIPTED). The simplest form of real-time 3D animation is quite common in games. The user has no control over scripted animations — they just play on regardless. These are almost always designed to repeat endlessly ("loop"), and thus are often rhythmic. Oil pumps churning away, animated Vegas neon signs, ventilation fans, and stoplights are simple examples. Scripting can be used to simulate complex actions, but it's not truly interactive. For example, take the "race yourself" feature in driving games. The player's animation during the previous lap is recorded, then connected to a competitor's car and played back. In this context, motion capture is form of scripted animation (more later on this).

TRIGGERED. When the user can launch a scripted animation, we call it "triggered." Simple examples include a "walk/don't walk" sign with a button, or an elevator door and button. Triggered animations also can be used in simple interactive designs, such as a trap-door triggered by a character's position.

SCRUBBED. I've never seen this used, but one could create an animation that is not meant to be played frame by sequential frame. Instead, frames are accessed at various speeds, even backwards and still-framed ("scrubbed" is the video-tape term for jumping through a tape). The direction, speed, and start/stop points are controlled directly at run time. Again, I'm not sure what you'd use it for, but it could be done.

USER INPUTS. Obviously, the user's input affects the scene, and usually it animates something. Most games are more complicated than simple "joystick-forward means move forward" — the user's input is filtered through physics simulation, calibrations, control mapping, and so on. No matter: in some form or another, the user drives the animation. Multiplayer games are great examples of the extremes of user-input.

ARTIFICIAL INTELLIGENCE (AI) INPUTS. This is a catch-all phrase for any type of animation that is calculated from a formula on the fly. Examples include an enemy AI jet fighter — instead of flying a preset path, the plane reacts according to a fight-or-flight formula. The formulas usually attempt to simulate an intelligent being and depend on inputs such as player proximity. Needless to say, good AI is rare, but when it's good, it's the best option because it's the most flexible and reactive to the game's actual situation. A lot of work is being done in this area for character animation, including SIGGRAPH research papers about things such as "Intelligent Actors: Automated Digital Character Animation." But it's not

easy. Fully simulated character motion is currently rare-to-nonexistent in commercial games.

Usually, though, one of these methods alone doesn't cut it. For more subtle, complex simulations, we combine them. For example, a user-controlled space ship will simultaneously bob (scripted), be struck by an enemy shot (AI), and move based upon user's input.

If we combine user input with scripted animation, we get something that I call branching-loop animation. This is the basis for most game characters' animations. The principal is simple: the artist builds a library of prescripted motions, each of which start and end at a known "neutral" position (more on this later). During game play, the appropriate action is chosen, usually based on user's input, and played. At the end of the loop, the player gives another input and another animation is chosen.

Branching loop animation is important to understand because it's so widely used in games. After we cover some more basics, we'll take a closer look at its strengths and weaknesses.

Performance

The good news is that RT3D animation performance may not be an issue for you — it depends on what you're animating. But you aren't going to be satisfied with a flip little answer like that, now are you? OK, here's the full scoop.

In modeling, we use face count as a basic unit to estimate real-time performance. We also have to worry that other things (such as texture size) may hurt our performance. Animation is another secondary factor.

It's reasonable to measure animation performance in transforms per frame (See the "Terms" sidebar). A low number of transforms per frame means it's not a big performance hit (and/or disk space use) to animate, and we can store more frames, which gives us incredibly long sequences of "canned" animation and/or smoother motion. Also, depending on the system, we can often use more faces for objects with a low transform per frame.

"Gee," you think excitedly. "It's all so easy. All I have to do is use a low number of transforms per frame. So, uh, how do I do that?" The number of transforms depends on the type of animation that you need. Let's take a look at the basic types:

Solid. The animating object doesn't move within itself. Bricks, spaceships, missiles, and cars are examples. Their animation data can be stored as a single transform per frame. Note that this doesn't rule out complicated motion (for example, a dogfight can be a realistic, complex animation), but it does require that the animated object animate as a single object (a bird couldn't flap its wings).

Jointed. With jointed models, we've divided the object into pieces, and we almost always connect them in a hierarchy as well. We're all familiar with human hierarchy, but it also applies to nonhuman objects. For example, a good car model would have independent motion for each wheel. For this type of animation, one transform for each object in the hierarchy, plus the data for the hierarchy itself. That's usually not very much at all since most hierarchies have less than 20 pieces, and the results often are really impressive.

Morphing. With morphing, we're way up the price/performance curve. Each vertex in the object can be separately transformed, which means there's a lot more effort for the computer.

Other. The "other" category for animation includes the odd stuff. Examples include animated textures, material animation, and 2D animation effects.

Terms You Ought to Know, Part 2

Frame. This term is used in two subtly different ways in RT3D. I'm sure you already know that in prerendered animation, a "frame" is a short, fixed length of time (often 1/30 sec) in a sequence, just as frame in a movie. Let's call this meaning "film-frame." The other usage is the basic performance yardstick , as in "frame rate" — it's how many times per second the computer draws the screen. The frame-rate's unit is frames per second (fps), and the name-droppers among us also use Hertz (Hz), a physics term that means cycles per second. These two meanings are different.

One of the places where these two terms collide is when we deal with the concept of real time. This is another term with two meanings: we game developers usually use it to mean interactive, as in "rendering-while-you-wait." This irritates simulation folks, because real time originally meant "synchronized to real-world clock time," such as when your computer reminds you of an appointment at 10:30. Let's call this second meaning real-world time.

Most game programmers, especially the performance-hacker sort, aren't concerned with keeping their game aligned with real-world time. The application's sense of time varies depending on what's on the screen. This becomes a wee problem when the artists create animation using the film-frame concept. The programmer has to write code to play the artist's animation, and that code doesn't know what 1/30 of a second is since it only knows frame rate. The result: animations that speed up or slow down with the frame rate.

For example, our horribly flawed game, Sex Among Turtles, has a crazy frame rate that varies between 10 and 60 fps. We artists deliver a 30-frame animation of the turtle sexily crawling through the sand, and we cheerfully explain that we planned for it to play at 30fps, so it's a 1-second animation. The programmer sort of looks down when we mention that, but since it's a horribly flawed project anyway, he gets our animation integrated and goes out for a stress-reducing bike ride. Imagine our surprise when we see the result: the turtles go scuttling across the sand like angry ants, bumbling along twice as quickly as we had planned. In the slow frame-rate areas, they look really cold-blooded as they lazily crawl at 1/3 their intended speed.

To solve it, the programmer would figure out the actual system time, then figure out what film-frame to show at run time. This will often result in suboptimal animation (skipped film-frames at low frame rates), but the timing will be corrected.

Of course, we can't really pin this whole problem on confusing terminology, but understanding terminology can sometimes be the difference between comprehension and confusion.

Whew, I feel better. Back to animation.

FIGURE 1. *Branching loop animation.*

Branching Loop Animation

Let's take a closer look at how branching-loop animations work with interactions (Figure 1). First, let's state the obvious: if we're scripting the character's motion, then the user doesn't have full, continuous control over the character's action.

This isn't good for game play or animation quality. What if the user changes the action during the playback of the sequence? For example, they push the jump button, and then the punch button a split-second later? There are many ways to resolve this (this is where playing other games can teach us a lot). Here are a few typical solutions:

All too often, the jump animation must finish before the punch begins. This is the easiest solution to code, and looks best during production, but it's not all good. For the artist, this solution means we have to build very short animation cycles (a major constraint) because the user is "locked in" to the action they choose.

A slight improvement is adding more direct control over the entire character movement. For example, the user can change the direction of the character's walk during the walk cycle. This is done by applying a single transform to the character while the animation plays. But the problem remains: to punch, they have to wait for the jump to finish.

Sometimes the user can abort mid-action — for example, half-way through a jump, just start punching. This is excellent for instant response to the user's input, but obviously it looks really weird when half-way through the air, the player instantly snaps back to the neutral position and throws a punch. In fact, this looks so bad that it's rarely used.

Another solution is to allow separate animations for different parts of the character. Torso animations play separately from leg animations, allowing a punch and a jump at the same time. This doesn't fully solve the problem, but it does allow for many animations to be combined. This option sounds good, but it's difficult to animate a torso if you don't know what the legs are going to be doing — the artist has to compromise the animation quality severely by making indistinct actions at the waist. It's also a major pain, organizationally, during development. This does work nicely for some areas, though: hand and face animations can easily be separated out from the rest of the body.

A more appealing option is blending animations. This works something like weighted morph targets, in which various animations are blended together depending on the user input. If our user selects a jump during the punch, the game attempts to combine the two actions. This approach is risky — often it works very smoothly, but because it's being calculated at run time, it's unpredictable and can look awful (body parts running through each other) if everything happens to be lined up wrong.

NEUTRAL POSITION. The neutral position chosen is important because it's seen so often. The player will see it at the start and end of every animation, so it should reflect the character's individuality as much as possible. We also need to consider all the actions that the character will take, and choose a reasonable compromise pose among them.

We also want to choose a pose that is close to the fastest loops. If our neutral pose has the arms outstretched crucifixion-style, a punch animation is

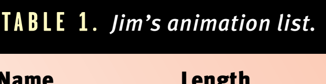


TABLE 1. *Jim's animation list.*

Name	Length	Start-pose	End-pose
Walk	15 frames	Neutral	Neutral
Nod	5 frames	Neutral	Neutral
Sit	35 frames	Neutral	Sit
Stand	35 frames	Sit	Neutral
Excited	40 frames	Neutral	Neutral

FIGURE 2. *Character Design Sketch*

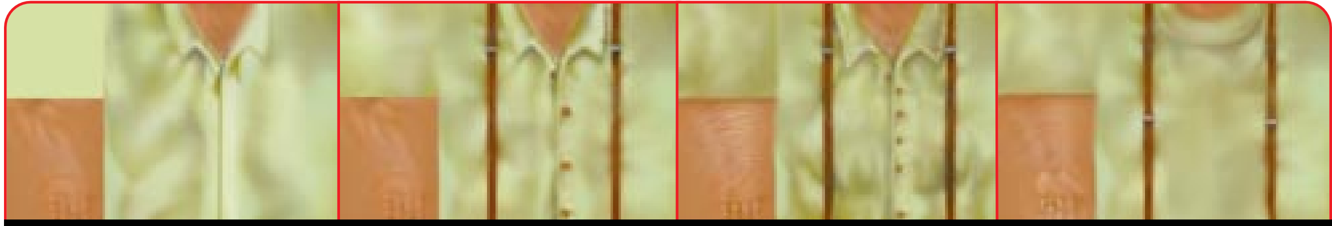


FIGURE 3. *(left to right) Basic colors with highlights; Suspenders and buttons, sleeve shadows; Final touches: arm hair, ring, more wrinkles; Back side of shirt.*

going to take a long time because we have to get the arms down. If we set the character in a boxer's stance, a quick jabbing punch can be animated realistically in 1/10 of a second or less.

Finishing Jim

Anyway, if you've joined us from last month, where we constructed the head of a real-time 3D character, you'll recall that we were planning to build the body this month. After looking at it, I decided there wasn't much to say about the modeling for the rest of the body — it's pretty straightforward in my opinion. If you feel stranded, write and let me know (better yet, write with a specific set of questions), and I'll cover it in a future column. So let's jump straight to the animation.

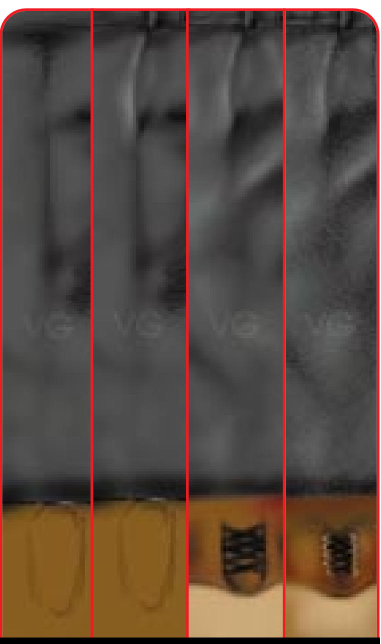


FIGURE 4. *(left to right) Basic colors with shadows; Highlights added; Basic shoe design, folds in pants; Noise to pants, button and shoe details.*

Now we'll summarize the completion of Jim's model (Figure 2). Painting the textures is an iterative 2D painting exercise — the images (painted by Lynell Jinks — see "Contributors") tell the story better than words can(Figure 3). Note that the arm included in the shirt texture is compressed slightly . This design keeps the textures rectangular, but by using careful mapping, we stretch out the arm to the correct length. We do lose a little resolution, but it's not very noticeable. Compared to using separate files for the arms and shirt, it allows the textures to be painted and edited quickly and easily, with fewer mistakes and better blending at the seams.

Next, we paint the lower area (Figure 4). The shoes are flattened out in the texture and then applied to the geometry using special mapping.

And now we have the hat (Figure 5). The hat is cylindrically mapped, which means that the texture isn't very intuitive to look at — it's the soup-can label concept.

With the textures complete, Lisa Washburn (see "Contributors") jumped into the 3D modeling. She built the rest of Jim's geometry, applying maps and linking (Figures 6 and 7)

Our finished model has 496 faces in 30 separate objects. Listing 1 shows data from 3D Studio MAX's Summary Info.

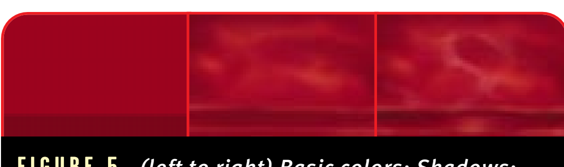


FIGURE 5. *(left to right) Basic colors; Shadows; Highlights.*

A close look will reveal some oddities in that list. First, each body part is named after its controlling joint. For example, "knee" is the lower leg section. This reminds us that we're animating joint rotations, not movement. Second, we've got two objects for every body part: `l_knee` and `l_knee01`. That's because our real-time animation system only allows one material per object, but most body parts need multiple textures: one for front, one for back. The solution (besides asking programmers for multiple materials per object, which doesn't work) is to create separate objects that are linked together.

LISTING 1. *Jim's Summary Info.*

```
Objects: 30
  Name (Type)            Verts    Faces
  ------------------     ------   ------
  hips (Mesh)            32       20
  r_hip (Mesh)           11       13
  r_knee (Mesh)          7        7
  r_ankle (Mesh)         15       22
  r_ankle01 (Mesh)       6        4
  r_knee01 (Mesh)        10       7
  r_hip01 (Mesh)         10       11
  hips01 (Mesh)          28       24
  l_hip (Mesh)           11       13
  l_hip01 (Mesh)         10       11
  l_knee (Mesh)          7        7
  l_ankle (Mesh)         15       22
  l_ankle01 (Mesh)       6        4
  l_knee01 (Mesh)        10       7
  torso (Mesh)           14       16
  head (Mesh)            74       112
  Hat (Mesh)             29       40
  torso_01 (Mesh)        16       20
  r_shoulder (Mesh)      11       14
  r_elbow (Mesh)         13       13
  r_wrist (Mesh)         20       22
  r_wrist01 (Mesh)       8        6
  r_elbow01 (Mesh)       10       7
  r_shoulder01 (Mesh)    7        6
  l_shoulder (Mesh)      11       14
  l_elbow (Mesh)         13       13
  l_elbow01 (Mesh)       10       7
  l_wrist (Mesh)         20       22
  l_wrist01 (Mesh)       8        6
  l_shoulder01 (Mesh)    7        6
```

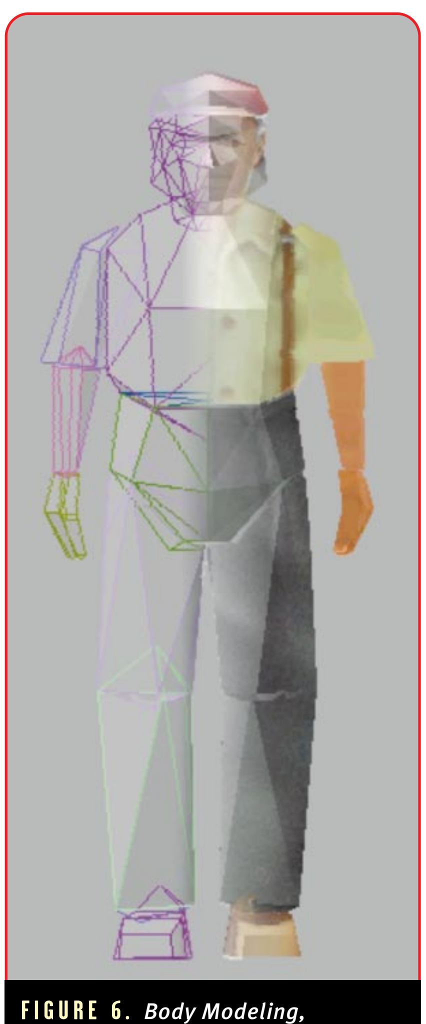


FIGURE 6. *Body Modeling, Mapping, and Linking*

Animation Design

Way back in the character design column, we learned that Jim isn't the lead character, so here's the decisions that were made for Jim:

Range of motion. A talk with the game designer told us that Jim doesn't need any complex animation — he's going to limp around and wave his arms a bit, but nothing too detailed.

Speech. The difficult decision about facial animation was made early on — they're using text in cartoon bubbles for dialog, so we won't animate speech. This will save lots of hard work doing lip-sync (obviously at the expense of realistic mouth motion, as the game designer is quick to point out).

Joints. Even though we could do "skinned" vertex animation, we'll be using intersecting joints. "This character plays a minor role, and its movement is relatively small, so intersecting joints probably won't look too bad," our art director said.

From this, we make decisions about the design of our character. Most of this brain action is about figuring out the best joints.

Joints. Since our joints overlap, we'll build them with vague textures in the intersecting areas, and model them so that the joint silhouettes look good during animation. We conclude that we'll build joints that look bad when flexed horribly far, in exchange for good-looking joints in small motions.

Next we'd have a meeting with the art director and game designer. We need a specific list of animated sequences, and we want to agree on a reasonable neutral position and targeted frame rate. In our example, our designer needs a shuffling walk cycle, a Yoda-like wise nod of approval, a sitting-down sequence, a standing-up sequence, and an excited gesticulation loop. We'll have the neutral position be the first frame in the nod animation, and we'll shoot for an animation speed of 15 frames per second. We'd also think about syncing these animations with any dialog. Once we'd agreed on these, we'd talk them through with the art director, acting them out and sketching poses for any confusing or difficult parts.

When we've finished, our animation list looks like Table 1.

Alas, animation decisions are often pushed to the end of the project, so artists don't know what animations their character is going to perform. In this case, it's wise to build an "exercise" animation, in which the character goes through the range of motions that are anticipated. This way, we can build joints that look good in the exercise animation and we'll have some kind of assurance that the final animations will also look good, as long as they don't exceed the bounds of the exercise animation. That means that the range of motion in the exercise animation is really important — it needs to be kept as small as possible because the larger the range of motion, the worse the joints are going to look.

Link-n-pivots. Linking and placing pivots is no different from normal prerendered animation. There are only a few differences or points to be aware of. For example, if you use your 3D modeling software to constrain the pivots, those settings often don't export properly.

For these kinds of issues, it's time to consult your programmer. Other good topics:

- Frame 0 transform (position, rotation, scale) of character.
- Neutral pose — does it matter to the programmer?
- Review of supported animatable transforms. For example, position? Rotation? Scale? Non-uniform scale? Other stuff (vertex color, morphing, and so on)?
- Standard names for body parts.

It's also a good time to review the way that you'll get your animation out of your modeling software and into the graphics engine. It's really helpful to sit with the programmer in front of your modeling software and point to an actual model.

For example, when our programmer sees our model, he says, "No problem, artist dude. All you do is save it out as a VRML 2.0 file and it'll load right in." I hope that it's obvious that the next step would be to generate a really quick animation test, and see if it

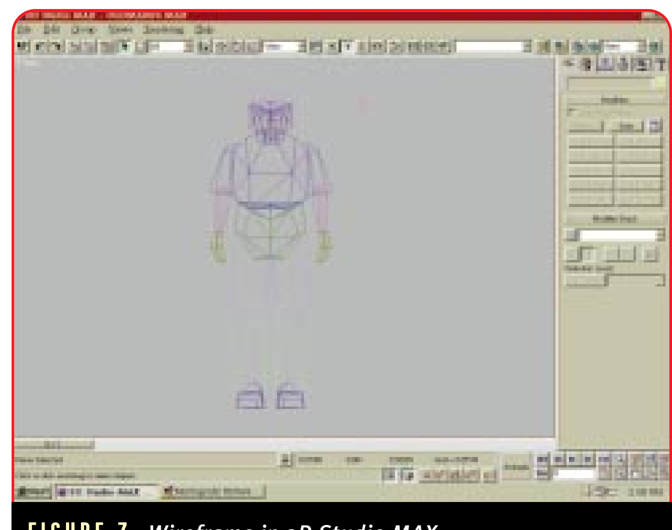


FIGURE 7. *Wireframe in 3D Studio MAX.*

FIGURE 8. *Completed animated character.*

speed up the development time. (Version 2, demonstrated at SIGGRAPH, will incorporate motion capture data, which I think will be a major improvement.)

MOTION CAPTURE DATA. If you're using motion capture data, you'll usually need to do some editing to make it loop, and you'll need to constrain the raw data so that it's expressed in the transforms that your graphics engine can handle. For example, raw motion capture data is often a series of XYZ points in space. You'll need some kind of tool to convert it into joint rotations within a hierarchy, and reduce the number of body parts to match your character (usually motion capture tracks more parts than real-time characters have).

There are a number of products that are designed to do this. I've used Motion Manager from BioVision, which does all those things within 3D Studio MAX. It works pretty well, though it does have its unique annoyances. There is one major problem: it uses custom-written 3D Studio MAX controllers, which means that the data is stored in its own unique transform (BioRotate, BioScale) instead of the standard 3D Studio MAX method. The result is that if you don't have Motion Manager, the character's transforms are effectively reset, leaving the body parts in a heap on the floor — not good at all. This isn't mentioned anywhere that I saw, and it's the sort of problem that can really hurt a production schedule. I asked BioVision about this problem, but haven't gotten an answer yet.

But there's another idea that didn't come through too well in these columns: character modeling is profound.

actually loaded right in. If it does, then you can generate your animation with confidence.

KEYFRAME ANIMATION. Once you have an animation list, building a keyframe animation is pretty straightforward — it's basically the same as "normal" 3D animation . You'll want to be careful to use only transforms that are supported in the real-time engine, which you can test by exporting your animation from time to time and loading it.

CHARACTER STUDIO. Kinetix developed a motion simulator plug-in for 3D Studio MAX that generates animations for characters. It's not perfect, but for some kinds of motion (especially fluid motions such as dancing), it's a really quick way to get good-looking animation data. It also has a decent library of existing motions, which really helps

Wrap-up

This is the end, finally, of the character-creation series. Reading back over these articles, I see that you, dear reader, can learn a lot from them. Still, compared to actually building characters for a real, live product, there's a lot more to learn. Interactive character modeling is the opposite of modular artwork: there's usually an endless array of annoyingly unpredictable problems that crop up. More than any other kind of modeling, there's no substitute for experience — I highly recommend that you give it a try (even if it's on your own time).

But there's another idea that didn't come through too well in these columns: character modeling is profound. Like this: Earlier this year, deep into late-night deadline stress, I was editing animation loops for this character. Fully immersed in the choppy sea of pivot-constraint-hierarchy-UV-weld technicalities, I looked up and saw this little guy strutting across my screen, like a shoemaker's elf. It was the animation I was building, but I was so far into the tech stuff that it actually surprised me.

I sat there thinking, I created something totally new and it's almost alive. I zinged off on one of those philosophical artistic tangents: What *is* it? It's so far from reality that we have to carefully simulate the dirty, gritty, messy stuff, and that's really different from natural media. It's not real in any sense, yet it feels more alive and realistic than a real, live ant. It's pretty darn close to pure human creativity, to raw life infused into abstract math (Figure 8).

That's it. As always, I need more feedback. Do me a favor and respond, good or bad, and I'll be very grateful. E-mail column@vectorg.com (or contact the magazine if you don't have e-mail) and let me know what you thought. ■

Contributors

Lisa Washburn is the lead RT3D artist at Vector Graphics. With her background in fine art, she uses sculpturing skills as well as her 3D modeling abilities to build magically delicious low-polygon models.

Lynell Jinks is a professional artist for Vector Graphics. He created the pencil sketches and textures shown here. His talent in 2D character artwork spans natural media as well as Photoshop texture and image creation.

HIRING GAME DESIGNERS

BY ARNOLD HENDRICK

In these enlightened days, most game developers and publishers have heard that a development team needs a "game designer." Some even know what a designer does. A game designer isn't necessarily the one dreaming up cool new game ideas. Game ideas/topics are often directed, and always approved, at the highest levels of management.

So what does a designer do? In short, a designer does a lot of writing: design documents, the user interface, goals of the game logic, dialogue and screen text, frequently the first draft of the manual, and sometimes the entire manual. A designer also researches data, provides algorithms or tables for certain parts of the game play, works with the team continually to refine and revise the game, and is a major participant in the play testing process.

Arnold Hendrick spent ten years designing paper wargames, RPGs, and miniatures rules before his 1982 arrival in computer games. Since then, he spent three years in the "cart game" trenches at Coleco, enjoyed MicroProse's ups and downs for ten years while working on various well-known products, and for the last two years has been involved in building and guiding the design staff of Interactive Magic.

Illustration by fillip yip

A-Z
X
4
+
5
6
÷
FiLiP

The greatest problem faced by companies employing designers is how to find and hire good ones. Almost anyone with the remotest connection to game development will tell you either (a) their real goal has always been to be a designer, or (b) they already are a designer because they did "some" design work on project X. Meanwhile, corporate executives trade horror stories about egotistical designers who rant and rave, kick Coke machines into junk, and start childish Usenet flame wars weekly.

On the opposite side of the fence, a small horde of potentially good designers is dying for a chance to break into the big leagues. They all wonder how to position themselves to be attractive to potential employers. The employers, meanwhile, wonder how to find the next genius among the hordes trying to storm the citadel.

Designers Come in Two Sizes

Game design work has two distinct levels: lead and assistant. The vision and game play decisions of the lead designer guide the game toward commercial success. Even if top management dictates the genre and topic, its directives rarely exceed a paragraph or two. Turning those brief paragraphs into a fun, money-making game is where the lead designers exercise their craft and creativity.

Some games require more design work than the lead designer can handle, especially if the schedule is tight or the project is large. Assistant designers are the ditch diggers who diligently work on those tiresome details that the lead designer lacks the time to accomplish. These details might include nitpicking research, setting up level maps, grinding out data tables, or scripting text blocks and voice-overs. In time, the assistant designers (and their employer) hope they'll learn more about making games; enough to permit their ascent from the trenches to the exalted status of lead designer.

This discussion deals with the quantifiable skills and background that an employer can evaluate when considering different candidates. It is assumed that anyone doing a competent job of hiring can evaluate prior experience and determine if a person is likely to fit into or clash with the corporate culture.

The First Cut: Literacy

The core skill of game design is the ability to write well. Designers must be able to write discursive, analytical prose that clearly communicates complicated concepts. It's amazing how many people lack this ability. Invariably, these people make poor designers. Their design documents will be a mess, in-game text will be confusing at best, and they are no help at all with the game manual. Besides, good writers are handy elsewhere. In a crisis, a literate designer could come up with a press release, web page text, or even box and ad copy. It might not be great, but it shouldn't be too embarrassing either.

Unless a candidate has obvious professional writing or editing experience, the best way to evaluate his or her ability is to examine a writing sample. Lead designers should be able to provide their previous game work. Assistant designers should have something that they've worked on, even if it was never published. Something game-oriented is naturally preferable. You should write off any applicant who can't show you a writing sample. Writing is a skill that must be practiced, and that practice inevitably produces something that you can read.

Another reason to demand good writing is that it's impossible to write well without a certain amount of intelligence, organization, and clear thinking. An inability to write may be the iceberg tip of far greater weaknesses.

Depth and Breadth of Knowledge

The designer is the central source of information about a game's topic. A topic-challenged designer may need months to read and research enough to become a semi-expert. A designer who is familiar with the subject can immediately start thinking about how subject and game play might converge.

For example, my current employer, Interactive Magic, publishes numerous contemporary and historical games with a military theme. We expect our designers to bring some background to this field and have fairly decent gaming experience within it. Some months ago, while interviewing a prospective designer, I asked him what era of military history or contemporary military affairs he understood best. We started talking about the classical era (Greeks and Romans), but it quickly became apparent that most "ancients" miniatures gamers had a better feel for that period. We tried WW II, where at least he could mention some famous pieces of equipment. Unfortunately, he couldn't describe what equipment opposed these famous pieces, or why these opponents were overmatched. I don't expect every designer to master every period, but a good designer needs to have dug into the details of at least one period or genre.

I also probe for breadth of knowledge. Designers are more effective if they understand something about graphic design, art, music, and theater. The best designers that I know are renaissance men and women with numerous interests and abilities.

Knowing Games

Good game designers keep up with games published in their field. It's impossible to play every game, but familiarity with a respectable variety, good and bad, helps one avoid past errors and profit from past successes. A game-challenged designer might need a month to find and play representative titles of the genre, and would still lack the extra insights that germinate during animated pro-and-con discussions about various games. Meanwhile, the knowledgeable designer can anticipate the thorny issues of game play and help steer a team away from dead-ends and toward useful answers.

When I interview prospective designers about game play, I always apply my professional/amateur acid test. This involves discussing various games that we both know, preferably games similar to the ones he or she will work on — although in a pinch, anything will do. We talk about what features we felt were successful and unsuccessful. We discuss how these features contributed to the overall success or failure of the game. A candidate who can talk only about what he or she enjoys, and has

no interest in the opinions or attitudes of others, fails the test. If they genuinely are interested in trying to figure out what gamers want, identifying what features seem to attract customers, and banishing elements that drive customers off, then they pass the test with flying colors.

A designer must go beyond personal preferences and try to understand what customers want. It's dumb luck if your personal preference happens to match that of the general public. This lucky match can happen once or twice. Unfortunately, it rarely lasts. I know a couple of egocentric designers who were very successful in the 1970s. Although their products from then are still known today, their subsequent work has passed unmarked by any success. Another example occurred a few years back when a well-known game designer "retired" because the public wasn't ready for and didn't appreciate his work.

Lead designers really must have a strong grasp of the genre in which they work. This often leads to specialization among designers. For example, one of the Interactive Magic design staff is famous for his attitude toward anything science fiction or fantasy: "Never touch the stuff!" Nevertheless, he happens to be one of the world's most experienced designers and inveterate players of nineteenth-century wargames, and he knows and plays the twentieth century just as well, not to mention ancients. For a company heavily involved in military strategy games, this designer is a priceless asset.

On the other hand, breadth of ability is important. Very often, staff designers get matched to projects simply because the designer happens to be free. The flexibility to do a good job in a field outside your specialty increases your job security and improves your chances of getting hired. My own resume includes published credits in simulation, strategy, RPG, and even console action games. This really helps when (not if) the company folds or you're caught in a downsizing.

The Importance of Cool Ideas

Innumerable people believe that they'd make a great game designer because they have a cool idea for a game. Unfortunately, because so many people have so many cool ideas, different people frequently come up with the same cool idea. Furthermore, most game companies spend at least 90 percent of their resources milking a past cool idea that "made it big," and less than 10 percent gambling on the next cool idea. When they do gamble, it's because people like the chairman of the board, the president, or some vice president insist that the company bet on their cool idea.

What a game designer contributes is the zillions of cool small ideas that make a game better, even if the president's cool idea actually is tired and lame. A good game designer will flesh it out, add some nifty features, downplay the irrational stuff, and deliver a product with some chance of success in the marketplace.

Anyone seriously interested in game design automatically has lots of cool ideas. Any designer worth his or her salt can give you three blockbuster ideas before breakfast. I know I could do it, but never has my employer asked, "Hey, make us a game for Christmas next year — your choice, just so long as it sells well." Still, I've been more fortunate than most. Exactly once during my 15 years in the industry, I was able to talk a company into doing "my idea."

Another indicator of a good designer is that he or she feels no obligation to be original. The real pros understand the value of reusing ideas that have worked in the past. Many people criticized DIABLO for being NET-HACK or ROGUE with cool graphics and sound. The truth is, NET-HACK and ROGUE were great games. The DIABLO team had the wisdom to take a well-proven idea and do it really, really well. A designer who ignores such lessons and insists on constant novelty is a financial disaster waiting to happen.

Teamwork

Today, games are created by teams of artists, programmers, designers, and increasingly a sound specialist. A good designer must work well with such a team. In both the interview and the reference check, be sure to probe for their attitudes towards others. An overweening ego almost invariably means poor teamwork skills. If a designer even hints at being an overbearing know-it-all who sneers at the rest of the world during an interview, don't expect him or her to suddenly become thoughtful, considerate, and collaborative with the development team.

On the other hand, a good designer, especially a lead designer, needs a certain amount of self-confidence and willpower to keep the game on a sensible path. Like any collaborative effort, games need a "direction giver." This person has the authority to prevent the effort from fragmenting into a mish-mash of features that pleases no one.

Technical Knowledge

Game designers need not be programmers. Even those who were once programmers find that being a good designer leaves them little time to code. However, a designer must have sufficient experience or native intelligence to understand what programmers and artists say. Lead designers need sufficient experience to know what should be easy, what will be difficult, and what is impossible. Every few years, a new tide of hardware and software washes through the industry. Designers need an awareness of this, since apparently miscellaneous bits of flotsam and jetsam can hold the keys to dramatic advances in game capabilities. Designers with recent work experience in large organizations have the advantage of strolling down the hall to get insights. The solo freelancer spends time and money discovering what is possible and what is not.

For example, I believe that the astute use of 3D art software (not 3D real-time display engines) to achieve animated, photorealistic scenes helped make COMMAND & CONQUER or DIABLO into megahits. Guessing right on programming protocols for 3D accelerators could be equally important for late 1998 and 1999.

Prior Experience

Naturally, experience in game software development is valuable. Prior experience should be a modifier to the factors mentioned previously. A

designer who seems to have the necessary abilities, insights, and attitudes will be more useful if he or she has experience. However, a designer with the appropriate qualifications but no experience is actually preferable to a veteran designer who can't write, has insufficient background, can't think analytically about games, is outrageously egocentric, and refuses to work on anything other than a current brainchild. Worse, a "poison pill" veteran will not only command a large salary, but will also need a big, expensive support staff to do all the real work. Hidden staff costs aside, I would always trade one "poison pill" designer for a brace of promising assistant designers.

Nevertheless, it's also risky to give an assistant designer a lead designer's job. Large companies, especially, benefit from at least one senior or lead designer to help the assistants along, guide their efforts, and nurture the best into lead designers. Naturally, being one of this sort myself, I believe companies should spend lavishly and wisely on this critical bit of senior talent. Still, in some cases, veteran lead designers need not be hired; at the moment, I know of numerous superbly qualified individuals who work as freelancers.

Recruiting

Finding good lead designers is very difficult. As with any professional position, a company is best served by a nationwide search, a willingness to examine agency candidates, and a general "rattling the network" to see who might be available and interested. Designers tend to know other designers, which makes networking exceptionally important.

Conversely, for assistant designers, companies are served best when they start close to home. Many good candidates may exist within the company, toiling away in play testing, customer service, or other junior positions. I've had the most luck with the play testing staff. Their continued presence proves that they can survive the horrors of finishing a game. More than once, I've invited play testers into a specific project on a probationary basis, just to see what they could do as an assistant designer. On occasion, I've been pleasantly surprised, and the person has gone on to a happy and successful career in design. Other times, I've seen my worst fears confirmed and had the unpleasant task of telling a person that their skills, abilities, and/or knowledge were insufficient to do the job.

Looking beyond the company itself, local universities and gaming groups can be talent gold mines. Even if you don't find any assistant designers, these people are often willing to work part-time in play testing. Ads in local newspapers can turn up some surprising candidates. One of the most successful "finds" at Interactive Magic was a meteorologist who just happened to have all the right skills and attitudes, despite a lack of professional experience. Within two years, he'd survived lead design challenges and moved up to an assistant producer role.

About "Breaking In"

Anyone seeking a first job in game design can infer much from this discussion. First, make sure you have the appropriate skills and can demonstrate them clearly to an employer. Some companies may have wacky ideas about game designers, but the level of intelligent hiring grows as the capitalistic equivalent of Darwinian selection bankrupts firms that consistently make poor decisions.

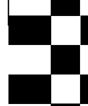
The best place to get a foot in the door is at a large firm that needs assistant designers. These companies are more likely to consider candidates with little or no experience. If a design job isn't available, consider a related position, perhaps in play testing. Even if you can't get promoted from within, a year or two of industry experience and product development exposure can help you snag an assistant designer position elsewhere. Another useful place to get experience is to volunteer your assistance to the various professional web sites that deal with gaming. Some marketing departments take these sites almost as seriously as print magazines; perhaps your interviewer will feel the same.

For those still making educational decisions, a four-year college degree at the most challenging school you can handle will help. A well-rounded liberal arts education can be as useful as math or computer science. It's easy to imagine courses that might help you write scripts for an introductory narration (public communications), research obscure historical data (history), guide a composer onto the right track (music appreciation), discuss screen layout and color with the lead artist (principles of design), understand the techniques and limitations of the new 3D engine (advanced algebra), then pitch in to write a decent manual (writing). Of course, some programming courses won't hurt either. Lack of a college degree need not be fatal, but those without a degree need work samples to prove that their abilities and skills are equivalent to a college education.

All companies hire in spurts. During the happy times when a company expands, they need people right away, if not yesterday. During the down times they just don't hire, period. Therefore, try to figure out which companies are doing well and check up with them frequently.

Work on your job hunting skills and apply them intelligently to the game industry. It always helps if you walk into an interview familiar with the company's products and future plans. That means playing their hits and recent releases, memorizing their announced list of future releases, and finding a way to reveal this knowledge in a cover letter or an interview. In interviews, always be careful with the classic question, "Give me an honest appraisal of our game X." Most people are testing not only your insight and honesty, but also your diplomacy. Congratulate them on what they did right, and offer suggestions for improvement in areas where they had trouble. A cardinal rule of business it to offer solutions, not problems. Find something nice to say about even their worst game and don't hesitate to point out weaknesses in competitive products.

Finding a job in game design can seem difficult to impossible. It requires patience and persistence to find a company that successfully filters out the clamor and concentrates on candidates who can really "do the job." ■



Ro a i g Objec U i g Q a e io

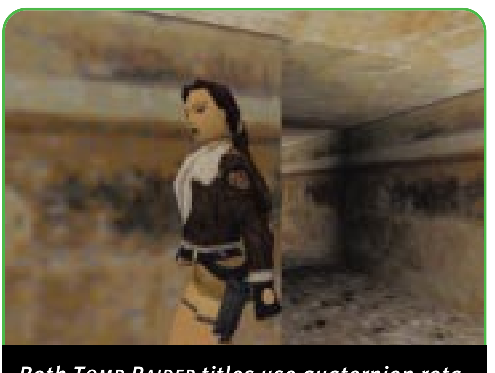
by Nick Bobick

Last year may go down in history as The Year of the Hardware Acceleration. Much of the work rasterizing and texture-mapping polygons was off-loaded to dedicated hardware. As a result, we game developers now have a lot of CPU cycles to spare for physics simulation and other features. Some of those extra cycles can be applied to tasks such as smoothing rotations and animations, thanks to quaternions.

Many game programmers have already discovered the wonderful world of quaternions and have started to use them extensively. Several third-person games, including both TOMB RAIDER titles, use quaternion rotations to animate all of their camera movements. Every third-person game has a virtual camera placed at some distance behind or to the side of the player's character. Because this camera goes through different motions (that is, through arcs of a different lengths) than the character, camera motion can appear unnatural and too "jerky" for the player to follow the action. This is one area where quaternions come to rescue.

Another common use for quaternions is in military and commercial flight simulators. Instead of manipulating a plane's orientation using three angles (roll, pitch, and yaw) representing rotations about the x, y, and z axes, respectively, it is much simpler to use a single quaternion.

Many games coming out this year will also feature real-world physics, allowing amazing game play and immersion. If you store orientations as quaternions, it is computationally less expensive to add angular velocities to quaternions than to matrices.



Both TOMB RAIDER titles use quaternion rotations to animate camera movements.

Interpolating the Orientation of an Object

There are many ways to represent the orientation of an object. Most programmers use 3×3 rotation matrices or three Euler angles to store this information. Each of these solutions works fine until you try to smoothly interpolate

Nick Bobick is a game developer at Caged Entertainment Inc. and he is currently working on a cool 3D game. He can be contacted at nb@netcom.ca. *The author would like to thank Ken Shoemake for his research and publications. Without him, this article would not have been possible.*

between two orientations of an object. Imagine an object that is not user controlled, but which simply rotates freely in space (for example, a revolving door). If you chose to store the door's orientations as rotation matrices or Euler angles, you'd find that smoothly interpolating between the rotation matrices' values would be computationally costly and certainly wouldn't appear as smooth to a player's eye as quaternion interpolation.

Trying to correct this problem using matrices or Euler angles, an animator might simply increase the number of predefined (keyed) orientations. However, one can never be sure how many such orientations are enough, since the games run at different frame rates on different computers, thereby affecting the smoothness of the rotation. This is a good time to use quaternions, a method that requires only two or three orientations to represent a simple rotation of an object, such as our revolving door. You can also dynamically adjust the number of interpolated positions to correspond to a particular frame rate.

Before we begin with quaternion theory and applications, let's look at how rotations can be represented. I'll touch upon methods such as rotation matrices, Euler angles, and axis and angle representations and explain their shortcomings and their relationships to quaternions. If you are not familiar with some of these techniques, I recommend picking up a graphics book and studying them.

FIGURE 1: *Euler angle representation.*

Using Rotation Matrices

To date, I haven't seen a single 3D graphics book that doesn't talk about rotations using 4×4 or 3×3 matrices. Therefore, I will assume that most game programmers are very familiar with this technique and I'll just comment on its shortcomings. I also highly recommend that you re-read Chris Hecker's article in the June 1997 issue of the *Game Developer* ("Physics, Part 4: The Third Dimension," pp.15-26), since it tackles the problem of orienting 3D objects.

Rotations involve only three degrees of freedom (DOF), around the x, y, and z coordinate axes. However, nine DOF (assuming 3×3 matrices) are required to constrain the rotation — clearly more than we need. Additionally, matrices are prone to "drifting," a situation that arises when one of the six constraints is violated and the matrix introduces rotations around an arbitrary axis. Combatting this problem requires keeping a matrix orthonormalized — making sure that it obeys constraints. However, doing so is not computationally cheap. A common way of solving matrix drift relies on the Gram-Schmidt algorithm for conversion of an arbitrary basis into an orthogonal basis. Using the Gram-Schmidt algorithm or calculating a correction matrix to solve matrix drifting can take a lot of CPU cycles, and it has to be done very often, even when using floating point math.

Another shortcoming of rotation matrices is that they are extremely hard to use for interpolating rotations between two orientations. The resulting interpolations are also visually very jerky, which simply is not acceptable in games any more.

Using Euler Angles

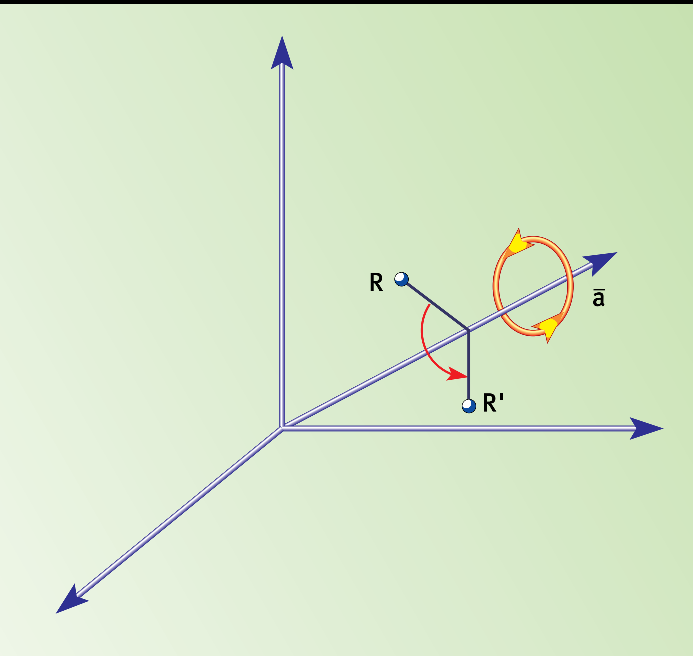
You can also use angles to represent rotations around three coordinate axes. You can write this as (θ, χ, ϕ); simply stated, "Transform a point by rotating it counterclockwise about the z axis by θ degrees, followed by a rotation about the y axis by χ degrees, followed by a rotation about the x axis by ϕ degrees." There are 12 different conventions that you can use to represent rotations using Euler angles, since you can use any combination of axes to represent rotations (XYZ, XYX, XYY...). We will assume the first convention (XYZ) for all of the presented examples. I will assume that all of the positive rotations are counterclockwise (Figure 1).

Euler angle representation is very efficient because it uses only three variables to represent three DOF. Euler angles also don't have to obey any constraints, so they're not prone to drifting and don't have to be readjusted.

However, there is no easy way to represent a single rotation with Euler angles that corresponds to a series of concatenated rotations. Furthermore, the smooth interpolation between two orientations involves numerical integration, which can be computationally expensive. Euler angles also introduce the problem of "Gimbal lock" or a loss of one degree of rotational freedom. Gimbal lock happens when a series of rotations at 90 degrees is performed; suddenly, the rotation doesn't occur due to the alignment of the axes.

For example, imagine that a series of rotations to be performed by a flight simulator. You specify the first rotation to be Θ^1 around the x axis, the second rotation to be 90 degrees

FIGURE 2. *Angle and axis representation.*



around the y axis, and Θ^3 to be the rotation around the z axis. If you perform specified rotations in succession, you will discover that Θ^3 rotation around the z axis has the same effect as the rotation around the initial x axis. The y axis rotation has caused the x and z axes to get aligned, and you have just lost a DOF because rotation around one axis is equivalent to opposite rotation around the other axis. I highly recommend *Advanced Animation and Rendering Techniques: Theory and Practice* by Alan and Mark Watt (Addison Wesley, 1992) for a detailed discussion of the Gimbal lock problem.

Using Axis and Angle

Using an axis and angle representation is another way of representing rotations. You specify an arbitrary axis and an angle (positive if in a counterclockwise direction), as illustrated in Figure 2.

Even though this is an efficient way of representing a rotation, it suffers from the same problems that I described for Euler angle representation (with the exception of the Gimbal lock problem).

The Quaternion Solution

In the eighteenth century, W. R. Hamilton devised quaternions as a four-dimensional extension to complex numbers. Soon after this, it was proven that quaternions could also represent rotations and orientations in three dimensions. There are several notations that we can use to represent quaternions. The two most popular notations are complex number notation (Eq. 1) and 4D vector notation (Eq. 2).

$$w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \text{ (where } i^2 = j^2 = k = -1 \text{ and } ij = k = -ij \text{ with real } w, x, y, z) \quad \text{(Eq. 1)}$$

$$[w, v] \text{ (where } v = (x, y, z) \text{ is called a "vector" and } w \text{ is called a "scalar")} \quad \text{(Eq. 2)}$$

I will use the second notation throughout this article. Now that you know how quaternions are represented, let's start with some basic operations that use them.

If **q** and **q´** are two orientations represented as quaternions, you can define the operations in Table 1 on these quaternions.

All other operations can be easily derived from these basic ones, and they are fully documented in the accompanying library, which you can find on the *Game Developer* web site. I will also only deal with unit quaternions. Each quaternion can be plotted in 4D space (since each quaternion is comprised of four parts), and this space is called quaternion space. Unit quaternions have the property that their magnitude is one and they form a subspace, S3, of the quaternion space. This subspace can be represented as a 4D sphere. (those that have a one-unit normal), since this reduces the number of necessary operations that you have to perform.

It is extremely important to note that only unit quaternions represent rotations, and you can assume that when I talk about quaternions, I'm talking about unit quaternions unless otherwise specified.

Since you've just seen how other methods represent rotations, let's see how we can specify rotations using quaternions. It can be proven (and the proof isn't that hard) that the rotation of a vector **v** by a unit quaternion **q** can be represented as

$$v' = q\, v\, q^{-1} \text{ (where } v = [0, v]) \quad \text{(Eq. 3)}$$

The result, a rotated vector **v´**, will always have a 0 scalar value for w (recall Eq. 2 earlier), so you can omit it from your computations.

TABLE 1. *Basic operations using quaternions..*

Addition:	**q** + **q´** = [w + w´, v + v´]
Multiplication:	**qq´** = [ww´ – v · v´, v **x** v´ + wv´ +w´v] (· is vector dot product and **x** is vector cross product); Note: **qq´** ≠ **q´q**
Conjugate:	**q*** = [w, -v]
Norm:	N(**q**) = $w^2 + x^2 + y^2 + z^2$
Inverse:	**q-1** = **q*** / N(**q**)
Unit Quaternion:	q is a unit quaternion if N(**q**)= 1 and then $\mathbf{q}^{-1}$ = **q***
Identity:	[1, (0, 0, 0)] (when involving multiplication) and [0, (0, 0, 0)] (when involving addition)

Conversions to and from Quaternions

Today's most widely supported APIs, Direct3D immediate mode (retained mode does have a limited set of quaternion rotations) and OpenGL, do not support quaternions directly. As a result, you have to convert quaternion orientations in order to pass this information to your favorite API. Both OpenGL and Direct3D give you ways to specify rotations as matrices, so a quaternion-to-matrix conversion routine is useful. Also, if you want to import scene information from a graphics package that doesn't store its rotations as a series of quaternions (such as NewTek's LightWave), you need a way to convert to and from quaternion space.

ANGLE AND AXIS. Converting from angle and axis notation to quaternion notation involves two trigonometric operations, as well as several multiplies and divisions. It can be represented as

$q = [\cos(\Theta/2), \sin(\Theta/2)\mathbf{v}]$ (where Θ is an angle and v is an axis)

(Eq. 4)

EULER ANGLES. Converting Euler angles into quaternions is a similar process — you just have to be careful that you perform the operations in the correct order. For example, let's say that a plane in a flight simulator first performs a yaw, then a pitch, and finally a roll. You can represent this combined quaternion rotation as

$q = q_{yaw}\, q_{pitch}\, q_{roll}$ where:

$$q_{roll} = [\cos(\psi/2), (\sin(\psi/2), 0, 0)]$$
$$q_{pitch} = [\cos(\theta/2), (0, \sin(\theta/2), 0)]$$
$$q_{yaw} = [\cos(\phi/2), (0, 0, \sin(\phi/2)]$$

(Eq. 5)

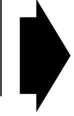
The order in which you perform the multiplications is important. Quaternion multiplication is not commutative (due to the vector cross product that's involved). In other words, changing the order in which you rotate an object around various axes can produce different resulting orientations, and therefore, the order is important.

ROTATION MATRIX. Converting from a rotation matrix to a quaternion representation is a bit more involved, and its implementation can be seen in Listing 1.

Conversion between a unit quaternion and a rotation matrix can be specified as

$$R_m = \begin{bmatrix} 1-2y^2-2x^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix}$$

(Eq. 6)



It's very difficult to specify a rotation directly using quaternions. It's best to store your character's or object's orientation as a Euler angle and convert it to quaternions before you start interpolating. It's much easier to increment rotation around an angle, after getting the user's input, using Euler angles (that is, roll = roll + 1), than to directly recalculate a quaternion.

Since converting between quaternions and rotation matrices and Euler angles is performed often, it's important to optimize the conversion process. Very fast conversion (involving only nine `mul`s) between a unit quaternion and a matrix is presented in Listing 2. Please note that the code assumes that a matrix is in a right-hand coordinate system and that matrix rotation is represented in a column major format (for example, OpenGL compatible).

LISTING 1: *Matrix to quaternion code.*

```
MatToQuat(float m[4][4], QUAT * quat)
{
  float  tr, s, q[4];
  int    i, j, k;

  int nxt[3] = {1, 2, 0};

  tr = m[0][0] + m[1][1] + m[2][2];

  // check the diagonal
  if (tr > 0.0) {
    s = sqrt (tr + 1.0);
    quat->w = s / 2.0;
    s = 0.5 / s;
    quat->x = (m[1][2] - m[2][1]) * s;
    quat->y = (m[2][0] - m[0][2]) * s;
    quat->z = (m[0][1] - m[1][0]) * s;
} else {
          // diagonal is negative
           i = 0;
        if (m[1][1] > m[0][0]) i = 1;
              if (m[2][2] > m[i][i]) i = 2;
          j = nxt[i];
          k = nxt[j];

          s = sqrt ((m[i][i] - (m[j][j] + m[k][k])) + 1.0);

              q[i] = s * 0.5;

          if (s != 0.0) s = 0.5 / s;

             q[3] = (m[j][k] - m[k][j]) * s;
          q[j] = (m[i][j] + m[j][i]) * s;
          q[k] = (m[i][k] + m[k][i]) * s;

           quat->x = q[0];
           quat->y = q[1];
           quat->z = q[2];
           quat->w = q[3];
  }
}
```


If you aren't dealing with unit quaternions, additional multiplications and a division are required. Euler angle to quaternion conversion can be coded as shown in Listing 3.

Achieving Smooth Interpolation

One of the most useful aspects of quaternions that we game programmers are concerned with is the fact that it's easy to interpolate between two quaternion orientations and achieve smooth animation. To demonstrate why this is so, let's look at an example using spherical rotations. Spherical quaternion interpolations follow the shortest path (arc) on a four-dimensional, unit quaternion sphere. Since 4D spheres are difficult to imagine, I'll use a 3D sphere (Figure 3) to help you visualize quaternion rotations and interpolations.

Let's assume that the initial orientation of a vector emanating from the center of the sphere can be represented by q1 and the final orientation of the vector is q3. The arc between q1 and q3 is the path that the interpolation would follow. Figure 3 also shows that if we have an intermediate position q2, the interpolation from q1 → q2 → q3 will not necessarily follow the same path as the q1 → q3 interpolation. The initial and final orientations are the same, but the arcs are not.

Quaternions simplify the calculations required when compositing rotations. For example, if you have two or more orientations represented as matrices, it is easy to combine them by multiplying two intermediate rotations.

$R = R_2R_1$ (rotation R_1 followed by a rotation R_2) (Eq. 7)

LISTING 2: *Quaternion-to-matrix conversion.*

```
QuatToMatrix(QUAT * quat, float m[4][4]){
  float wx, wy, wz, xx, yy, yz, xy, xz, zz, x2, y2, z2;

  // calculate coefficients
  x2 = quat->x + quat->x; y2 = quat->y + quat->y;
  z2 = quat->z + quat->z;
  xx = quat->x * x2;   xy = quat->x * y2;   xz = quat->x * z2;
  yy = quat->y * y2;   yz = quat->y * z2;   zz = quat->z * z2;
  wx = quat->w * x2;   wy = quat->w * y2;   wz = quat->w * z2;

  m[0][0] = 1.0 - (yy + zz);   m[0][1] = xy - wz;
  m[0][2] = xz + wy;           m[0][3] = 0.0;

  m[1][0] = xy + wz;           m[1][1] = 1.0 - (xx + zz);
  m[1][2] = yz - wx;           m[1][3] = 0.0;

  m[2][0] = xz - wy;           m[2][1] = yz + wx;
  m[2][2] = 1.0 - (xx + yy);          m[2][3] = 0.0;

  m[3][0] = 0;                        m[3][1] = 0;
  m[3][2] = 0;                        m[3][3] = 1;
}
```

This composition involves 27 multiplications and 18 additions, assuming 3×3 matrices. On the other hand, a quaternion composition can be represented as

$q = q_2 q_1$ (rotation q_1 followed by a rotation q_2)

(Eq. 8)

As you can see, the quaternion method is analogous to the matrix composition. However, the quaternion method requires only eight multiplications and four divides (Listing 4), so compositing quaternions is computationally cheap compared to matrix composition. Savings such as this are especially important when working with hierarchical object representations and inverse kinematics.

Now that you have an efficient multiplication routine, see how can you interpolate between two quaternion rotations along the shortest arc. Spherical Linear intERPolation (SLERP) achieves this and can be written as

$$SLERP(p,q,t) = \frac{p\sin((1-t)\theta) + q\sin(t\theta)}{\sin(\theta)}$$

(Eq. 9)

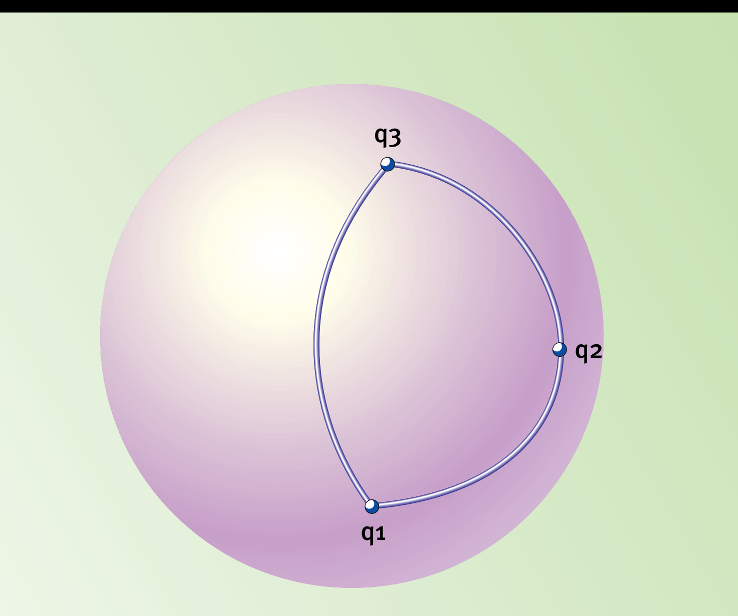
where $pq = \cos(\theta)$ and parameter t goes from 0 to 1. The implementation of this equation is presented in Listing 5. If two orientations are too close, you can use linear interpolation to avoid any divisions by zero.

The basic SLERP rotation algorithm is shown in Listing 6. Note that you have to be careful that your quaternion represents an *absolute* and not a *relative* rotation. You can think of a relative rotation as a rotation from the previous (intermediate) orientation and an absolute rotation as the rotation from the initial orientation. This becomes clearer if you think of the q2 quaternion orientation in Figure 3 as a relative rotation, since it moved with respect to the q1 orientation. To get an absolute rotation of a given quaternion, you can just multiply the current relative orientation by a previous absolute one. The initial orientation of an object can be represented as a multiplication identity [1, (0, 0, 0)]. This means that the first orientation is always an absolute one, because

$$q = q_{identity}\, q$$

(Eq. 10)

FIGURE 3. *Quaternion rotations.*



LISTING 3: *Euler-to-quaternion conversion.*

```
EulerToQuat(float roll, float pitch, float yaw, QUAT * quat)
{
        float cr, cp, cy, sr, sp, sy, cpcy, spsy;

// calculate trig identities
cr = cos(roll/2);
        cp = cos(pitch/2);
        cy = cos(yaw/2);

        sr = sin(roll/2);
        sp = sin(pitch/2);
        sy = sin(yaw/2);

        cpcy = cp * cy;
        spsy = sp * sy;

        quat->w = cr * cpcy + sr * spsy;
        quat->x = sr * cpcy - cr * spsy;
        quat->y = cr * sp * cy + sr * cp * sy;
        quat->z = cr * cp * sy - sr * sp * cy;
}
```

LISTING 4: *Efficient quaternion multiplication.*

```
QuatMul(QUAT *q1, QUAT *q2, QUAT *res){

float A, B, C, D, E, F, G, H;

A = (q1->w + q1->x)(q2->w + q2->x);
B = (q1->z - q1->y)(q2->y - q2->z);
C = (q1->x - q1->w)(q2->y - q2->z);
D = (q1->y + q1->z)(q2->x - q2->w);
E = (q1->x + q1->z)(q2->x + q2->y);
F = (q1->x - q1->z)( q2->x - q2->y);
G = (q1->w + q1->y)(q2->w - q2->z);
H = (q1->w - q1->y)(q2->w + q2->z);

res->w = B + (-E - F + G + H) /2;
res->x = A - (E + F + G + H)/2;
res->y = -C + (E - F + G - H)/2;
res->z = - D + (E - F - G + H)/2;
}
```

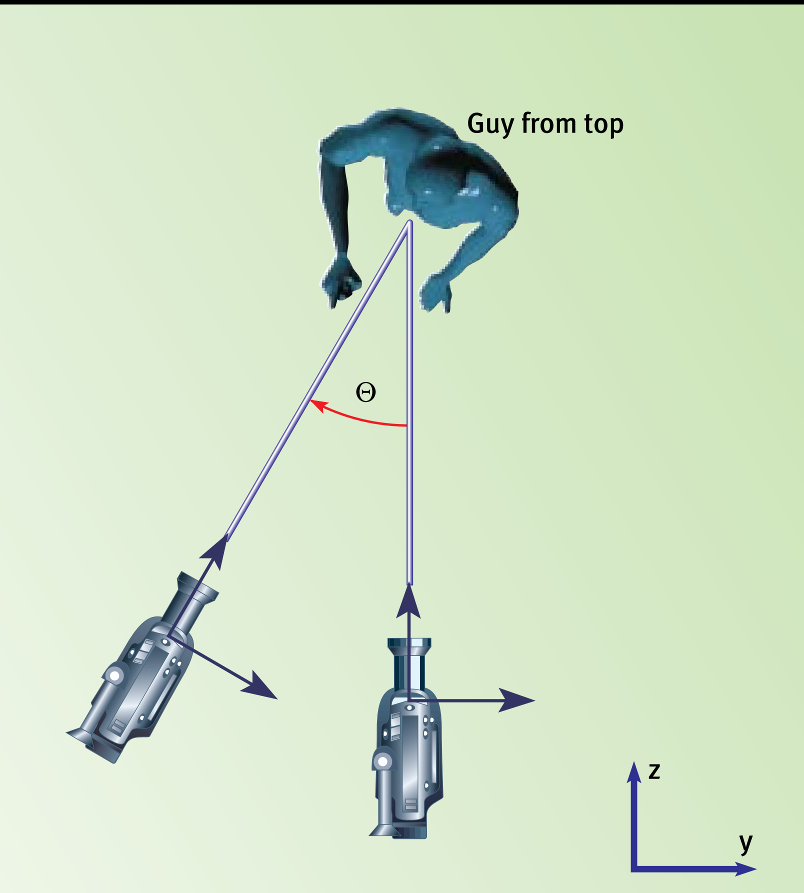


FIGURE 5. *Camera from top.*

Camera Implementation

As I stated earlier, a practical use for quaternions involves camera rotations in third-person-perspective games. Ever since I saw the camera implementation in TOMB RAIDER, I've wanted to implement something similar. So let's implement a third-person camera (Figure 4).

To start off, let's create a camera that is always positioned above the head of our character and that points at a spot that is always slightly above the character's head. The camera is also positioned *d* units behind our main character. We can also implement it so that we can vary the roll (angle θ in Figure 4) by rotating around the x axis.

As soon as a player changes the orientation of the character, you rotate the character instantly and use SLERP to reorient the camera behind the character (Figure 5). This has the dual benefit of providing smooth camera rotations and making players feel as though the game responded instantly to their input.

You can set the camera's center of rotation (pivot point) as the center of the object it is tracking. This allows you to piggyback on the calculations that the game already makes when the character moves within the game world.

Note that I do not recommend using quaternion interpolation for first-person action games since these games typically require instant response to player actions, and SLERP does take time. However, we can use it for some special scenes. For instance, assume that you're writing a tank simulation. Every tank has a scope or similar targeting mechanism, and you'd like to simulate it as realistically as possible. The scoping mechanism and the tank's barrel are controlled by a series of motors that players control. Depending on the zoom power of the

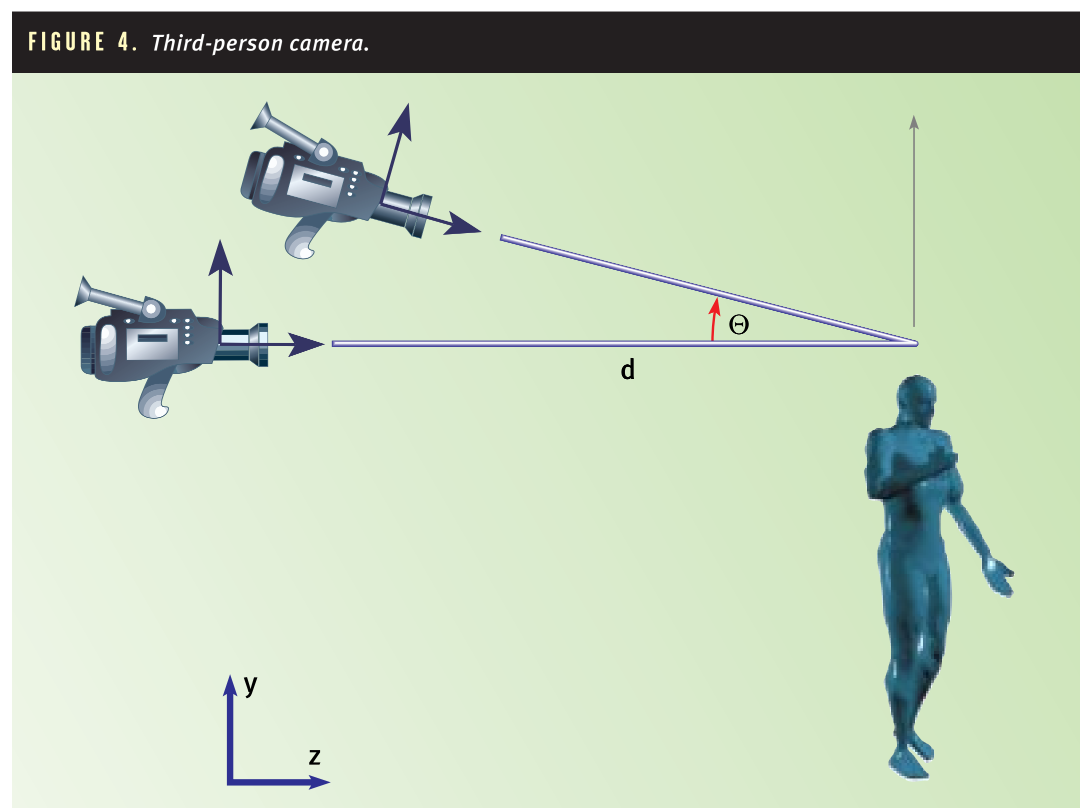


FIGURE 4. *Third-person camera.*

scope and the distance to a target object, even a small movement of a motor could cause a large change in the viewing angle, resulting in a series of huge, seemingly disconnected jumps between individual frames. To eliminate this unwanted effect, you could interpolate the orientation according to the zoom and distance of object. This type of interpolation between two positions over several frames helps dampen the rapid movement and keeps players from becoming disoriented.

Another useful application of quaternions is for prerecorded (but not prerendered) animations. Instead of recording camera movements by playing the game (as many games do today), you could prerecord camera movements and rotations using a commercial package such as Softimage 3D or 3D Studio MAX. Then, using an SDK, export all of the keyframed camera/object quaternion rotations. This would save both space and rendering time. Then you could just play the keyframed camera motions whenever the script calls for cinematic scenes.

LISTING 5: *SLERP implementation.*

```
QuatSlerp(QUAT * from, QUAT * to, float t, QUAT * res)
{
        float           to1[4];
        double        omega, cosom, sinom, scale0, scale1;

        // calc cosine
        cosom = from->x * to->x + from->y * to->y + from->z * to->z
                                 + from->w * to->w;

        // adjust signs (if necessary)
        if ( cosom < 0.0 ){
                  cosom = -cosom;
                  to1[0] = - to->x;
                  to1[1] = - to->y;
                  to1[2] = - to->z;
                  to1[3] = - to->w;
        } else  {
                  to1[0] = to->x;
                  to1[1] = to->y;
                  to1[2] = to->z;
                  to1[3] = to->w;
        }

        // calculate coefficients

       if ( (1.0 - cosom) > DELTA ) {
              // standard case (slerp)
              omega = acos(cosom);
              sinom = sin(omega);
              scale0 = sin((1.0 - t) * omega) / sinom;
              scale1 = sin(t * omega) / sinom;

        } else {
    // "from" and "to" quaternions are very close
             //  ... so we can do a linear interpolation
              scale0 = 1.0 - t;
              scale1 = t;
        }
          // calculate final values
          res->x = scale0 * from->x + scale1 * to1[0];
          res->y = scale0 * from->y + scale1 * to1[1];
          res->z = scale0 * from->z + scale1 * to1[2];
          res->w = scale0 * from->w + scale1 * to1[3];
}
```


Angular Velocity and Quaternions

After reading Chris Hecker's columns on physics last year, I wanted to add angular velocity to a game engine on which I was working. Chris dealt mainly with matrix math, and because I wanted to eliminate quaternion-to-matrix and matrix-to-quaternion conversions (since our game engine is based on quaternion math), I did some research and found out that it is easy to add angular velocity (represented as a vector) to a quaternion orientation. The solution (Eq. 11) can be represented as a differential equation.

$$\frac{dQ}{dt} + 0.5 * \text{quat}(\text{angular}) * Q \qquad \text{(Eq. 11)}$$

where quat(angular) is a quaternion with a zero scalar part (that is, $w = 0$) and a vector part equal to the angular velocity vector. Q is our original quaternion orientation.

To integrate the above equation ($Q + dQ/dt$), I recommend using the Runge-Kutta order four method. If you are using matrices, the Runge-Kutta order five method achieves better results within a game. (The Runge-Kutta method is a way of integrating differential equations. A complete description of the method can be found in any elementary numerical algorithm book, such as *Numerical Recipes in C*. It has a complete section devoted to numerical, differential integration.) For a complete derivation of angular velocity integration, consult Dave Baraff's SIGGRAPH tutorials.

Quaternions can be a very efficient and extremely useful method of storing and performing rotations, and they offer many advantages over other methods. Unfortunately, they are also impossible to visualize and completely unintuitive. However, if you use quaternions to represent rotations internally, and use some other method (for example, angle-axis or Euler angles) as an immediate representation, you won't have to visualize them. ■

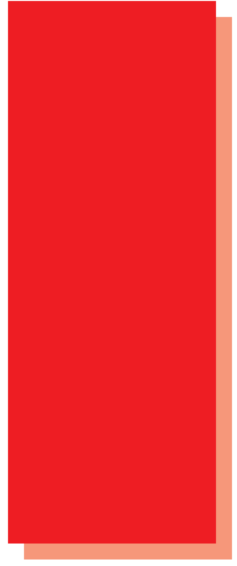
FOR FURTHER INFO

Numerical Recipes in C
http://cfatab.harvard.edu/nr/nronline.html

Dave Baraff's SIGGRAPH Tutorials
http://www.cs.cmu.edu/afs/cs.cmu.edu/ project/anim/aw/15-860/public/860home.html.

The Role of S o Bible i Game

by John Scott Lewinski



n the beginning, there were games. And the games were good. There were also stories. And they were good. But games and stories lived separately from one another, each satisfied to fulfill their lonely roles in the entertainment universe. So, stories and games each stood apart, refusing to contemplate fully the magic they could make together.... Until designers came along and said, “Let there be interactive games!” And they saw that it was good.

Outside of the entertainment industries, the term “bible” refers to the Good Book in its various regional forms. However, to professional storytellers, a bible is the blueprint for a plot. It’s a single publication that documents a story’s plot, characters, settings, and possible future developments.

Television and motion picture productions invented the concept of a story bible. That concept has now become commonplace in those arms of the entertainment industry. Working with the head writer of a television show or movie, a producer develops a bible for use by everyone creatively associated with the production. For example, every writer who writes a series episode or a story concept gets a copy of the bible. Costume designers, set decorators, animators, visual effects engineers, and many others all get a copy of the same document so everyone is literally on the same page throughout the production.

In game development, a bible systematically catalogs the plot-lines, characters, character histories, settings, and potential future stories for the game. Once in place, the bible serves as a blueprint or a template for every aspect of the produced story. If it isn’t in the bible, it doesn’t happen in the finished product.

The bible also can include a game narrative’s “back story” — the story that happened before the game that the player never sees. Why tell a story that no consumer will ever witness? Because that story can arise during the interactive game’s narrative. It can motivate a fight between characters because one holds a long-standing grudge for a rival. It can explain why the medieval town we visit in the game is half-burned down by the time we arrive. Most importantly, a complete back story can inspire the imaginations of designers and writers to add something special to the game.

Bibles typically play a role in the game industry’s more story-intensive titles — those that marry cinematic aesthetics, rich character development, and narrative. While there are plenty of great products that involve little or no story (such as fighting

John Scott Lewinski writes interactive and linear screenplays out of Los Angeles. He recently worked on the screenplay for COMMAND & CONQUER II: TIBERIAN SUN, the sequel to Virgin Interactive’s million-selling CD-ROM, COMMAND & CONQUER. He also co-wrote COMMAND & CONQUER, RED ALERT. In 1997, he contributed to the productions of RIVEN and THE JOURNEYMAN PROJECT III: LEGACY OF TIME. He holds a BA in Journalism from Marquette University and a Master of Fine Arts in Screenwriting from Loyola Marymount University. Contact him at jburrows@aol.com.

games or flight simulators), many influential games over the last five years, such as RIVEN, the 11TH HOUR, DARK FORCES, and THE DIG, used a plot to draw players deeper into the game universe.

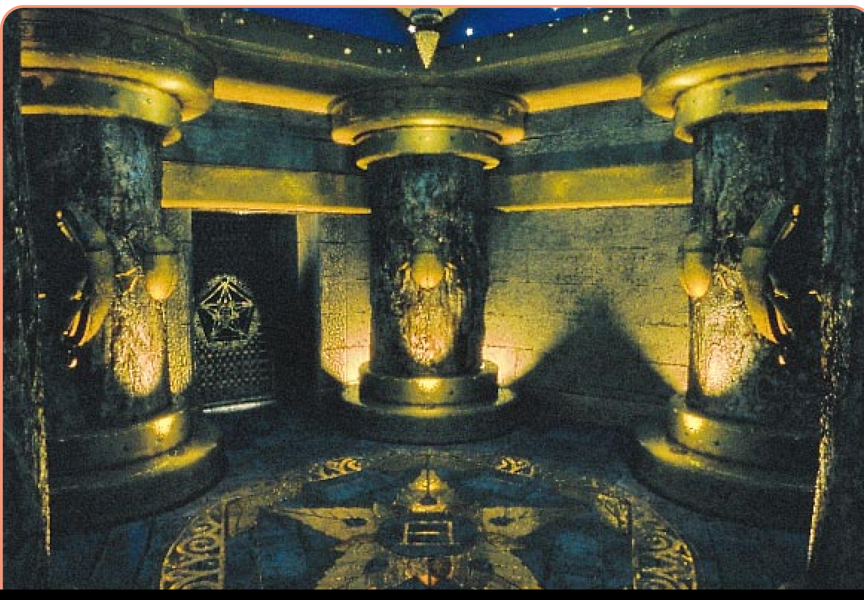
Why Use a Bible at All?

The dangers of working without a story bible can weaken an interactive product. Without a story bible, a member of the creative team might develop an idea or story element that is incompatible with the producer's or head writer's overall vision of the game. The document adds rich depth and detail to the game world and characters, so that appearances and behaviors are clear to writers, designers, and 3D artists.

Could you imagine the disaster if the writers, designers, and artists hadn't had a clear, all-around vision of Lara Croft in TOMB RAIDER? If the game writer envisioned her tough, no-nonsense personae correctly, but the 3D artist imagined the main character as a wispy Kate Moss clone, it would immediately be "back to the digital drawing board." In this example, such a simple change to Lara Croft could easily be communicated between developer, writer, artist, and designer. However, in games with many characters and locations (such as WING COMMANDER IV or THE JOURNEYMAN PROJECT), the role of a story bible becomes more important.

The story bible for a visually rich game such as RIVEN will focus on settings and venues.

A story bible can be included in a game's overall design document, or it can stand alone. The document typically runs from 20 to 50 pages long. Its composition should begin within the first weeks of a game's development (if story will be involved in the game). If a given game's design team includes a writer, it naturally falls to that scribe to compose the bible with the input and approval of the product's producer and designer. Any sort of interactive screenplay shouldn't so much as begin until the bible is set.

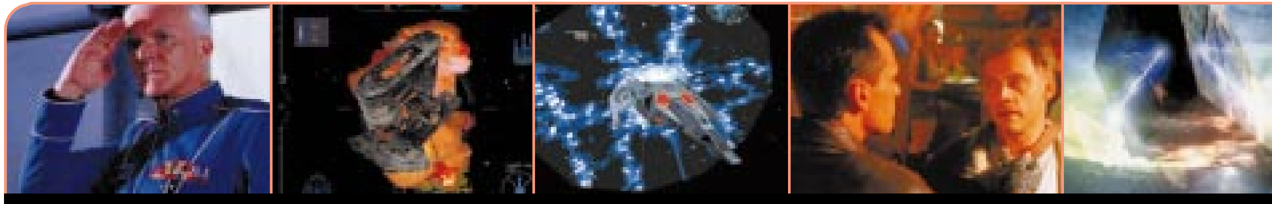


RIVEN's complex game play almost requires the organizational influence of a story bible.

How to Use a Story Bible

The overall development of game narrative should not continue until the story bible is completed, locked, and distributed. Once everyone is working from the same starting point, all creative members will find the going easier, with questions regarding character, setting, and such essentially answered. A caveat: While the bible should be agreed upon and completed before a game's narrative elements really gather steam, there is no reason why the bible can't be "unlocked" to add new characters, plot twists, or other brainstorms. If the game narrative gets bigger or fancier, the bible must be expanded to reflect that development.

Since game designers are often more at ease with the interactive entertainment genre than writers, and writers can generally handle the ins and outs of plot creation with greater ease than an experienced game designer, it makes sense that these two entities should work together to create the game bible.



In games with many characters and locations, such as WING COMMANDER IV, the role of a story bible is vitally important.

Just ten years ago, fleshing out the plot, characters, and other game details in writing was not so important to games. Many games relied on flashy graphics and maybe a little back story in the instruction manual. Now, many games are interactive narrative adventures. Some (such as Origin's WING COMMANDER IV, Activision's SPYCRAFT, and Virgin's THE 11TH HOUR) have merged with film aesthetics to create truly interactive motion pictures.

The creative process of interactive writing for games resembles that of television more than that of motion pictures. In movie screenwriting, the writer often works alone, whether he or she is writing the original script or doctoring a previous draft from another writer. In television, writers are usually members of a writing staff, collaborating with other staff writers and a script supervisor. Those writers all need to have the same information and identical understanding of plot, characters, and setting. What would a hero do? What wouldn't he do? Where can a story go? Where can it never go? The rules that are needed to answer these questions exist in the bible. Every writer has the same one, and every writer knows it cover to cover.

This can prove especially useful if a game title might include plans for future installments, add-on missions, or complete sequels. The concepts for those possible follow-ups can be installed directly into the bible. Then, the narrative of the original interactive game in question can include references to or a complete set-up for that proposed sequel. In general, the bible really takes a lot of doubt and potential narrative problems out of an interactive story line by answering many story questions.

Note that the bible is not the design document. The bible deals exclusively with story and its elements. While the design document guides the creation of the entire gaming experience, the bible controls the game's interactive screenplay.

And that's exactly how an interactive story is told — in screenplay or script form. Since most interactive games with story elements rely on cinematic aesthetics, it's only natural that the story be written in a format most similar to a movie or TV script (as opposed to prose style). Actors (even 3D rendered ones) have to be told what to say, and cameras need to know where to shoot. The screenplay puts all that in clear, formatted order.

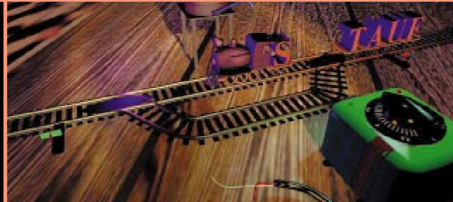
While interactive screenplay format can change from product to product depending on the genre, the role of the screenplay remains more or less the same. For example, THE DIG's screenplay was most likely dialogue intensive. A more visual game such as RIVEN could spend its words describing setting and venues. Also, an interactive screenplay usually runs significantly longer than a film or TV script. While a movie script runs no more than 120 pages, and a TV script no more than 60 (roughly a page per minute), an interactive script can run into the hundreds of pages. A screenplay for any interactive product needs to include story branching (with all the different scenes that players might encounter based on the choices they make) and a multitude of endings that players can enjoy depending on how well they play the game.

The interactive script can certainly interact with or even rest inside the final design document, since any effective interactive product needs to integrate design and screenplay successfully and skillfully. The game should drive the story, and the story should feed the game.

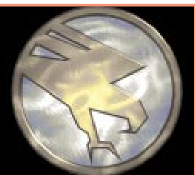
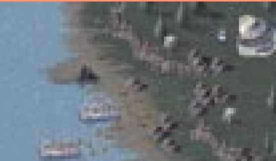
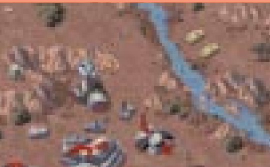
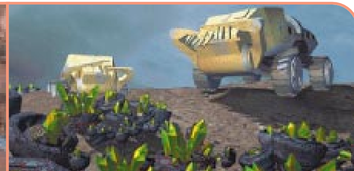
A Case Study of COMMAND & CONQUER II: TIBERIAN SUN

My experience with game bibles spans five different games (including work on RIVEN and the upcoming COMMAND & CONQUER II: TIBERIAN SUN) over the last three years. As a writer, I've contributed my storytelling skills to producers and developers who call on writers like me to help build a narrative in and around their game concepts. While a professional game designer might wrestle with missions, maps, and tile sets, I invent act breaks (how the story unfolds and where it twists or turns) and character biographies.

In the earliest development stages of Westwood's COMMAND & CONQUER II: TIBERIAN SUN, the initial design team of producer, designer, and writer (myself) collaborated on the development of such a bible. We discussed the characters and storyline of the origi-



11TH HOUR's game play depended heavily upon an involved plot that drew in players

In the initial stages of development of Command & Conquer II: Tiberian Sun, an effort was made to clarify and expand upon the C&C universe.

nal game, COMMAND & CONQUER, and its successful prequel, COMMAND & CONQUER: RED ALERT. We then decided how the game universe should progress in the storyline of C&C II. What resulted from this collaboration was the C&C II bible, which included some overview material about the COMMAND & CONQUER game universe, but focused primarily on the world of C&C II.

Once this bible came together, the C&C II design team was expanded to include artists, assistant designers, and animators. At this point, the bible really came into play. It could be handed to any new team member to bring them up to speed immediately on the details of C&C II and the overarching ground rules of the C&C universe. Those who read the C&C II bible all the way through had no questions about game's plot, characters, period, setting, or technology.

Following those initial discussions, we brainstormed about the new game's plot, characters, and setting. Over a series of meetings, we developed all the ingredients that would eventually go into the bible for COMMAND & CONQUER II: TIBERIAN SUN: new characters, old characters with new twists, the state of the game universe following the original C&C game, new settings, and so on. It then became my job to assimilate these ideas into some semblance of order and distill them into one document, the game's story bible. This bible would have to serve at a moment's notice should anyone new be brought in on the project and need to get up to speed on the game story, the characters, the characters' history, the games settings, the game's history, and so on.

Suffice it to say, C&C II's producer and lead designer made the major contributions to the initial drafts of the product's bible. The document grew slowly over the course of a few weeks. As a series of meetings clarified C&C II's overall history, the game's story, possible multiple endings, settings, character histories, back story, and all the other elements discussed so far came together into one document. That story bible was then absorbed into the much larger and more technical design document. Together, they were distributed to any creative team member that wanted to know the game's overall concept in total.

Unfortunately, I can't go into detail what the COMMAND & CONQUER II: TIBERIAN SUN bible contained or include any portion of it within an article. In fact, I can't reveal any tidbits on C&C II without ruining the surprise for all future players (and without getting sued for violating confidentiality agreements). So instead, I'll demonstrate the construction of a bible for a completely fictional product without stealing any details from the Virgin vaults.

A Fictional Example Of a Bible

Let's say we're working on a game titled HANGNAIL, the latest game inspired by QUAKE. HANGNAIL's bible would include a "treatment" or synopsis of the game's story.

LOG LINE. That treatment should include one- or two-sentence reviews of the story's beginning, middle, and end. In some cases, the treatment could go into greater detail, stretching from one page to twenty or more, if the designer or game writer chose to really flesh out the story in the design stage. If the game's narrative is truly based on cinematic story construction, the story might include first, second, and third act reviews. Leave those bits to your writer — we waste hours worrying about that act-structure nonsense. At the very least, the synopsis should include a "log line," or a brief review of the game's story (see the "HANGNAIL" sidebar).

CHARACTERS. The second portion of the

HANGNAIL

SYNOPSIS: A big, tough guy with heaps of muscles and a heart of gold walks through mazes and kills lots of stuff to battle evil, find his boxed lunch, and save the future of humanity... at least until the sequel comes out.

CHARACTER NAME: Dirk Squarejaw

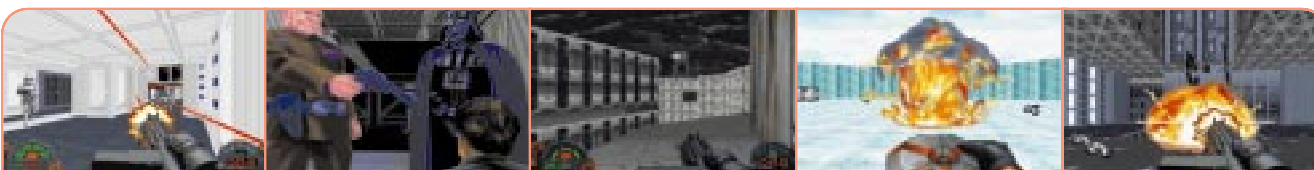
Age: Late 20s

Appearance: Ruggedly handsome and in the kind of impossibly good shape that you'd need to spend 25 hours a day in a gym to achieve.

Equipment: Death Ray of Death, Grenade of Severe Owies, Swiss Army Knife of Animosity, Pulse Cannon of Mild Mood Swings.

Attributes: Wonderfully and relentlessly violent, with an overdeveloped sense of honor; dedicated to saving all life on Earth, or at least all attractive women on Earth; enjoys painting in splattered blood, rainy days, long walks on the beach, thermonuclear devices, and backgammon.

Background: Orphaned at birth and raised by wolves, Dirk was rescued by nuns at the age of four. The nuns instilled in the young Dirk his sense of honor and his bizarre obsession with backgammon. When the evil villain General Payne destroyed the nuns' village to hijack all their dice, Dirk set out on his life-long quest to end evil around the world. He will never rest until Payne is defeated, peace and justice restored, and double sixes rolled everywhere.



DARK FORCES, which is based on the Star Wars universe, requires a story bible to keep many diverse plot, setting, and character details in order.

bible would include character reviews. The most important component of any effective narrative, whether it's in a game, movie, TV show, or novel, is good characters. They should have well-rounded histories and solid motivations. Most importantly, they should be clearly drawn out so anyone who reads the bible or works on the game sees the same person in their minds. If a writer or designer creates a game revolving around an Schwartzenegger-type action hero and fails to describe his all-American, psychopathic personality, the artist or renderer could end up drawing Marv Albert. The "HANGNAIL" sidebar illustrates what our character bible would say about HANGNAIL's protagonist.

All the information in the character description above could be distilled into one long paragraph entry, if the designer chooses to limit the length or the scope of the bible. However, every character in the game (even supporting players) should be presented in this same detail.

Such enriching character sketches can provide inspiration when planning game maps or missions (depending on the game's genre). For example, in HANGNAIL's case, given Dirk's devotion to backgammon, the designer could construct a maze or a level in which the objective is to slaughter all of General's Payne's agents to recover their ill-gotten dice.

Character description and background is one area where a story bible can really enrich an interactive game. If the bible can draw out a game's central character with convincing depth and detail, the production can present an interesting and exciting person around which you can build a game and story.

In some cases, the player becomes that character. In other games, the player merely guides an already existing character. In either case, the story bible can outline what the main characters wants. That's the key. The entire game story should be built around what the main character or hero wants and needs (be it the damsel in distress, a magic amulet, or the enemy capital). Anything that makes the game more entertaining — battles in the cold reaches of space, races through monster-filled mazes, or puzzle-solving through a haunted library — can stand between the hero and the goal. But, the goal must be clear, ever-present, and motivated. The story bible can help a design team do that.

In another example, if Dirk was scared of water because his wolf parents couldn't swim, the designer might wish to create and underwater level and cause Dirk's air supply to disappear quickly because he hyperventilates too easily.

Using a methodology such as this, in which you define the background, attributes, age, appearance, and equipment of a character, can help ensure truly motivated and enjoyable characters and give the design team ideas for game play. A game's characters need to be compelling. If the player becomes a hero in the game, that hero must be attractive enough that the player wants to assume that persona. A game villain should be rotten enough that the player generates genuine passion and satisfaction from defeating him or her.

An essential rule of thumb states that every character, even the most incredibly butch of heroes, needs to have weaknesses or shortcomings. If a character seems too omnipotent and has every skill imaginable down pat, no player will believe that he or she could possibly lose or die. You don't have to make your hero or heroine a simpering wimp, but don't make them invulnerable. Even Superman has his kryptonite.

In the final document, Dirk's bible entry might include an artist's sketch (if created early in the game development process) or a 3D rendering (if created farther along in the development process), which might also be the actual avatar used in the game — if the product makes it that far along.

To digress for just a moment, thus far I have approached the use of game bibles for story development solely from the perspective of the hero. Lately, games such as Bullfrog's DUNGEON KEEPER and LucasArts' DARK FORCES II have made it possible for players to assume the role of the villain. However, that doesn't turn the narrative rule on its ear — the same guidelines still apply. A villain also has wants and needs. In the best possible scenario, the bad guy wants exactly the same thing as the hero. In drama and writing courses, that's called the "Law of Conflicting Need." A good story (and therefore, a good

Bullfrog's DUNGEON KEEPER allows players to be the bad guy. Still, a story is a story, and this reversal of roles shouldn't significantly affect your approach to narrative.

game, if it has story components) has a protagonist and an antagonist wanting the same thing for perfectly opposite reasons. We usually want the hero to get to that goal before the villain. However, in games where we become the villain, we assume the motivations of the villain. The bible should outline the history, personality and motivation of the bad guy as well as the hero. That way, if we become the antagonist in game play, it works just as well if we had chosen the hero's role.

SETTING. The next segment of a properly constructed game bible should include a review of game settings — the actual locales that the characters will inhabit as the game plays out. Interesting, well-detailed venues can often inspire game-play elements.

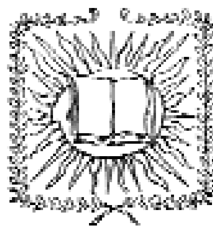
Setting includes everything from physical location to era to weather, climate, and so on. The setting description should also include any key items or environments that players will encounter. For example, if the climax of our imaginary game takes place in General Payne's Backgammon Emporium of Doom, we need to know that the setting includes giant, sentient radioactive dice that can leap from the walls and bulldoze our hero at a moment's notice.

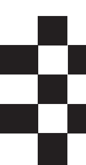
You don't want to go into too much detail to describe aspects of a setting that will never actually appear or play a part in the game. But the more thorough your setting descriptions in the story bible, the more fodder you provide for your eager game designer's information.

Preparing for a Sequel

Finally, a bible can include suggestions for future storylines. Television-series bibles that are given to writers who script episodes usually include one or two paragraph synopses of every show planned for that season. Likewise, a game bible can install storylines for games sequels, supplemental mission or level CD-ROMs, online additions, or spin-off products (such as comic books, novels, or TV shows).

While not every game needs a bible to provide addictive interactive entertainment, games that aspire to a more fully-developed and professionally presented storyline should consider the development of such a document. As more and more developers reach out for that cinematic style and aesthetic in their cut-scene look and game story, developers can rest assured that the procedures already exist out there amongst competent writers to construct the essential elements of a competent game narrative. ■





Ma ch Se : Ne fi e Ca al & To ch

by Dan Teven

Remember the first rule of game development: A game engine is only as good as the content that plays through it. The bargain bins are littered with games that had cutting-edge technology, but just weren't fun. Newfire is betting that developers would rather spend their time on game play than on optimizing texture mapping algorithms in assembly.

Torch is Newfire's game engine and Catalyst is Torch's authoring environment. Chances are, you'll decide whether or not to buy this suite of tools by evaluating Torch's price and performance, possibly comparing it to the likes of id Software's Quake engine — but there's a lot more to the product than Torch alone.

Catalyst's features for assembling and optimizing 3D worlds demonstrate Newfire's commitment to games, not just rendering technology. Catalyst doesn't force you to make too many difficult decisions. Your art department can create assets in standard formats, using the tools with which they're most comfortable, and Catalyst lets you integrate them smoothly into the world. Your programmers can define custom behaviors for objects, or they can take advantage of a fairly rich palette of standard behaviors. You can even choose to deploy your game via the Internet, rather than CD-ROM.

I happen to know a lot of programmers who live for optimizing assembly language, but Catalyst has something to offer them, too. TorchView is a window that monitors Torch's playback performance in real time, including statistics on CPU, rasterization, and texture usage. There's enough data available to satisfy the most diehard performance hacker.

Dan Teven is a consultant with 13 years experience in systems software development, games, and technical marketing. His all-time high Scrabble score is 554. He can be reached at dteven@ici.net.

The Newfire Architecture

Catalyst and Torch use VRML 2.0 to represent 3D worlds. (But note that Newfire has defined a few extensions to the VRML specification, and there are a few node types that it doesn't support yet.) Torch renders directly from VRML code, and Catalyst can generate VRML that's pre-optimized for fast playback. You can use the two pieces separately, but you'll only get the benefits of the optimizations when you use both together.

Torch is the most flexible 3D rendering engine I've seen. Torch can be deployed as a plug-in to a web browser, as a standalone player, or as an embedded component of an application (if you make special arrangements with Newfire). It's designed to be 3D hardware independent, but it exposes the advanced capabilities of the hardware

that it's running on. It supports rasterization in software (using either DirectDraw or Windows DIBs) and hardware (using 3Dfx's Glide or OpenGL, with Direct3D support in development). Newfire has a rasterizer development kit, so even more choices may be available soon.

Catalyst is not a full-featured VRML world editor, but it is powerful enough for prototyping or fine tuning existing worlds. Newfire expects you to create your models in another program, such as 3D Studio MAX, and import them into Catalyst, along with your texture maps, animations, and sound files. Catalyst can handle most standard asset formats, including .JPG, .PNG, and .GIF bitmaps, so it will work with your favorite authoring tools.

You can use features of VRML to control your game flow, if you wish. The VRML 2.0 language specification includes objects that generate and respond to events. Catalyst lets you insert these objects into your worlds and connect them in interesting ways, without writing a line of code. See the sidebar "VRML 2.0" for more information about creating interactivity through VRML.

For performance, flexibility, and familiarity reasons, Newfire recommends that you create your game logic in Java rather than VRML. You can add Java classes (created with a third-party Java programming environment) to your world with Catalyst. You can also control Torch directly from Java, through a published API.

You could also take a hybrid approach and use Java to write (for instance) your physics engine and user interface, but VRML to automate simple animated objects. This is the pragmatic way to go, since Java code takes time to create and debug. You might start with a prototype that's 75 percent VRML, tune it for a while, and end up with a product that's 75 percent Java.

Although Java is a wonderful language in which to program, Newfire's dependence upon it presents some problems. First, Java code doesn't execute as fast as C code. Sure, processors are still getting faster, and Java technology is improving, but game developers have always found reasons to push the pedal to the metal. If you're trying to find spare CPU cycles for a really advanced computer opponent, the Java performance penalty will hurt. On the other hand, if your application's going to be bound by rendering speed, the speed with which your Java code executes won't be an issue.

Second, Java is supposed to be a platform-independent language. Microsoft dearly wants you to be able to call the Windows API (and DirectX) from Java code, because doing so will tie your application to the Windows platform. Newfire seems to be firmly in the Microsoft camp — Torch requires Microsoft's Java Virtual Machine and doesn't run on any non-Windows platform — so you should be able to pull off this feat with Microsoft's JDirect. The flip side is that your game won't be Pure Java, and it won't run on a Macintosh (or WebTV, for that matter).

Third, although Java has built-in network support, it's pretty low-level. You'll still need to do a lot of work, or use a third-party library, to make a multiplayer game. The standalone Torch player doesn't have network support, so if you're designing a multiplayer game you should plan on delivering it within a browser window.

Finally, Java is still an uncommon choice for game development, and you might not find the breadth of library and tool support that exists for C or C++. Newfire plans to support C in the near future — maybe even before this article appears in print.

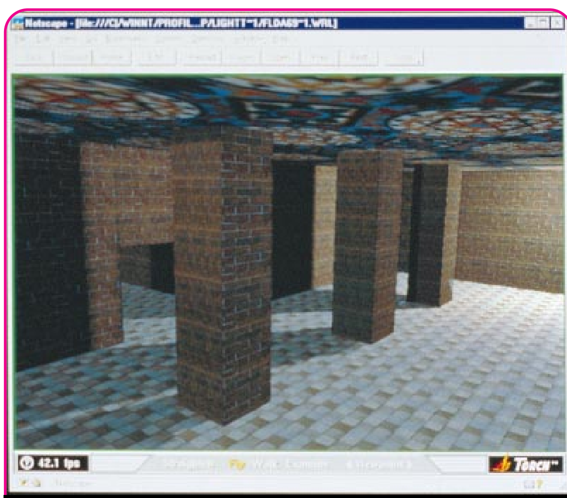
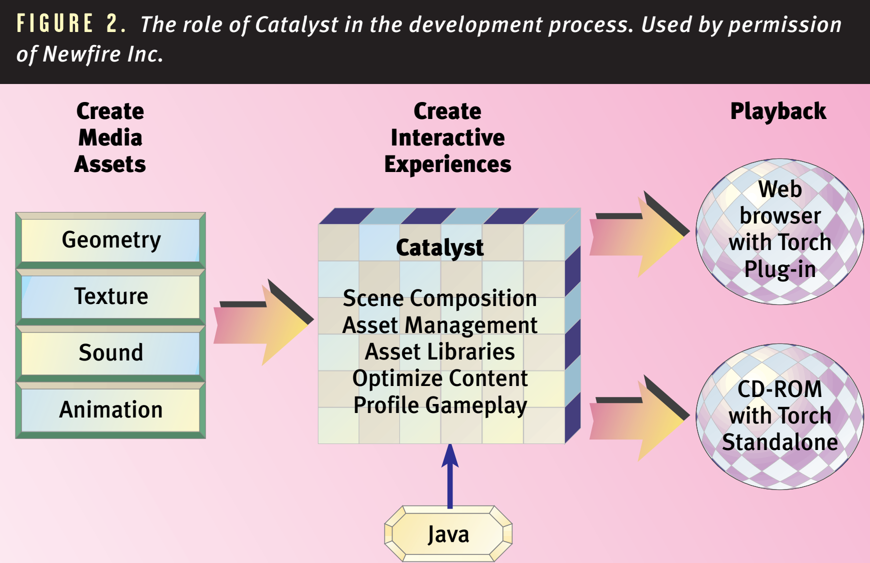


FIGURE 1. *Torch uses lighting and shadow maps to create impressively real environments.*

Torch Game Engine

Torch's rendering capabilities read like someone's notes from a Michael Abrash talk on the QUAKE engine: six degrees of freedom, scene management using a hybrid of Z-buffer and binary space partition (BSP) techniques, automatic MIP-mapping, distance-based level-of-detail model substitution, dynamic lighting for nontextured objects, and lighting

FIGURE 2. *The role of Catalyst in the development process. Used by permission of Newfire Inc.*



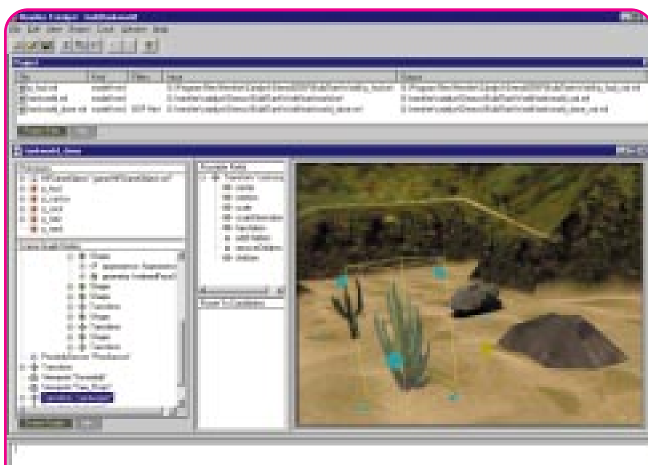


FIGURE 3. *Catalyst is a powerful VRML editor, but it still has some rough edges.*

maps. There are a few minor limitations, such as being able to render into only one window at a time, and allowing just one color palette per world (for 8-bit software rendering).

According to Newfire, Torch is designed to render 640×480 at 15 frames per second in software, or 30 frames per second with hardware acceleration, with effects such as lighting, shadows, and transparency. Torch didn't seem as fast as QUAKE to me, but ratchet the system requirements up a notch and it's definitely capable of rendering impressive-looking scenes at game speeds. I'd recommend targeting a 200 MHz Pentium if you're planning to use software rendering.

To take advantage of BSP culling, which is the key to fast playback, you need to use Catalyst to apply the "BSP hint" to a VRML world. Catalyst will use a set of heuristics to determine which objects should be included in the BSP tree (typically large, regularly shaped, unmoving objects) and which should be Z-buffered. You can view Catalyst's decisions and override them if necessary.

You can create lighting and shadow maps for a scene in the same way, by applying a hint within Catalyst. Since Torch can't light textured surfaces dynamically, this is a processor-efficient "cheat" that looks almost as good.

Torch handles textures quite well. It sets aside a 4MB texture cache, and if the textures required by a scene won't all fit, Torch resizes them temporarily. The software renderer can handle nonsquare textures in arbitrary sizes; the hardware renderers use the hardware's texture memory management, and may be limited. Textures can be transparent, and it's easy to animate them for stunning effects.

Torch can do all the important types of collision detection (object-to-object, object-to-scene, camera- to-scene, camera-to-object) and returns hit data at a level low enough to be compatible with your own physics engine. There are no facilities for higher-level physics management.

Sound support is basic, but includes both ambient and spatialized sounds (part of the VRML 2.0 specification). Torch currently does its own sound spatialization, but Newfire expects to switch to DirectSound3D in the future. There's support for single-channel MIDI, but not Redbook audio, even for CD-only titles. On the plus side, Torch supports all DirectInput devices.

When I started this review, I was worried that Torch would force game developers to develop games that all looked and felt similar. For the most part, it avoids this trap. The combination of BSP and Z-buffer rendering works well on a wide range of virtual environments, and you'll have to develop your own code for networking, physics, and artificial intelligence. Of course, if you're shopping for a game engine that handles 3D rendering with six degrees of freedom, you're probably not making a Scrabble game.

VRML 2.0

VRML (Virtual Reality Modeling Language) 2.0 is a relatively new specification, and most game developers probably aren't familiar with it. It's an interpreted language that describes a hierarchy of three-dimensional objects and their properties. For example, here's a code fragment that describes a red cube:

```
Shape {
  appearance
    Appearance {
      material
        Material {
          diffuseColor 1 0 0
          ambientIntensity 0.2
          shininess 0.2
        }
    }
  geometry
    Box {
      size 1 1 1
    }
}
```

This code fragment describes the cube using a hierarchy of four VRML nodes. The top-level `Shape` node has two exposed fields, `appearance` and `geometry`; think of these as the fundamental properties of a `Shape` object. Fields can refer to additional nodes, as `geometry` refers to a `Box` node, or they can define primitive properties, as `Box` defines the size of the cube.

In addition to basic geometry, VRML 2.0 lets you define texture maps, backdrops, elevation grids, fog effects, spatialized sound effects, and more. You can create sensor nodes that signal events when they detect the passage of time, the proximity or visibility of an object, or user input; these events can be routed to other nodes, which respond to them. For example, keyframe animation can be accomplished by routing a `TimeSensor` node to a `PositionInterpolator`.

Selecting VRML 2.0 for 3D world management has some disadvantages, but in general, it's a pretty good move. It's an Internet standard, so every content creation tool worth its salt will work with it. Of course, if you already have models defined in some other format, you'll have to convert them. The only real beef that I have is that VRML is an interpreted language, so VRML files have to be parsed every time they're read — hardly the fastest way to load world geometry.

Catalyst Development Environment

Catalyst looks something like Microsoft Developer Studio, with multiple windows that can float freely or can dock to other windows, local menus, and multilevel undo. I love Developer Studio's user interface, but Catalyst doesn't do as good a job at sizing and positioning the windows that I want on the screen at the same time — I found myself constantly wishing for a bigger screen. Also, once I found a layout that I liked, Catalyst didn't automatically save it.

In addition to the project window where you assemble the component files of your world, the most important windows are the scene graph window, which shows a tree view of the VRML geometry, and a window displaying a camera's view into the world. You'll probably spend most of your time in these two windows, cutting and pasting within the scene graph or directly manipulating objects in the world. There are other windows for searching the VRML tree and for managing the routings between VRML nodes.

You can have multiple worlds open at one time, each with their own sets of windows. However, I found Catalyst's behavior with multiple worlds open to be somewhat counterintuitive. If you undock your scene graph windows, their title bars don't contain the world name, so you can't tell to which world they belong. And if you cut something from World A's scene graph and then close World A, you can no longer paste it into World B — cut objects disappear from the clipboard.

One more gripe: to create a routing, you have to select something in two separate panes, and the command that you need is available in the local menu from just one of the panes.

My experience with Catalyst was productive, but small flaws such as these were typical. Catalyst is usable software that needs to go through another revision before it becomes really transparent to use. I did find some things I really liked about the interface. Previewing a series of level-of-detail models is painless: just zoom in or out from the object and watch closely as the geometry changes. Also, the dialog box where you assign commands to different buttons of different input devices is one of the cleanest examples of its genre that I've come across. I wish all games handled input device selection so well.

FIGURE 4. *TorchView's performance monitoring is an indispensable feature.*

TorchView

Officially part of Catalyst, TorchView is a wonderful feature. Getting a game to play back as fast as possible — and without any sudden dips in frame rate — has always been one of the most laborious parts of the development process. I've had to build execution profiling and real-time performance monitoring into more than one game, and it sure is nice to come across a game engine with these tools built in.

TorchView makes it absolutely painless to track statistics such as frame rate, polygon count, texture-memory use, CPU utilization, and the amount of time spent in each segment of the rendering pipeline. Not only can you see the raw numbers in real time, but you can chart them to get an idea of how they change while you navigate through a scene.

Fit and Finish

Installation was simple, meaning there is no option for full or partial installation and things went more or less according to formula. I was surprised by Setup's attempt to add the Torch .DLL directory to my DOS PATH. (For those readers too young to remember, this is the way we used to install software before the Windows registry was invented.)

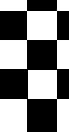
I found about the right number of bugs for a 1.0 release, mostly small ones. Newfire responds to bug reports promptly, so the 1.0 bugs should be ancient history by the time you read this review.

The documentation is a mixed bag, and it's not very well organized or indexed. The manual is supplemented by a thick binder called the Ignition Toolkit, by several code snippets and demos, and by material on Newfire's web site. The binder has good tips on

designing your game, and the manual covers the nuts and bolts of using Catalyst. I'd have liked a lot more information on delivering a finished product, on using Catalyst in a group environment, and on the rendering pipeline. The on-line help is HTML based, and updates are available from the web site.

Is It Worth the Money?

Newfire's Catalyst suite is an ambitious, well-designed product that will get better with age. The $1,995 price tag for the development kit is reasonable. And though you'll have to pay royalties of at least $1 a copy when you ship your game, the fee structure frees you to develop a game before you have a publishing deal. Furthermore, since Newfire gets rich only if your game's a huge hit, you can be confident they'll do whatever they can to help. If I were facing a build-or-buy decision today, I'd go the buy route — and I'd give Torch and Catalyst very serious consideration. ■



Catalyst 1.0/Torch 1.0

Newfire Inc.
Saratoga, CA
408 996-3100
www.newfire.com

Price: $1,995 per seat. Also royalties, payable when your game ships.

Software Requirements: Windows 95 or Windows NT 4.0; Netscape 3.0 or Internet Explorer 3.0; DirectX 3a and Microsoft Java Virtual Machine (provided); Java programming environment recommended.

Catalyst Hardware Requirements: Pentium 166, 64MB RAM, 50MB hard disk; Pentium Pro, 3D card recommended.

Torch Hardware Requirements: Pentium 133, 16MB RAM, 2MB VRAM; Pentium 166, 32MB RAM, 3D card recommended.

Technical Support: Installation support free for 30 days, development support $495/year.

Return Policy: 30-day

Rating (out of five stars):

Pros:

1. Torch is a fast, feature-rich rendering engine.
2. Newfire has done a nice job of creating a flexible, forward-thinking architecture.
3. Facilities for analyzing and improving game playback speed are built in.

Cons:

1. System requirements are high for both development and playback.
2. Release 1.0 only supports Java.
3. Both the documentation and the Catalyst user interface need more refinement.

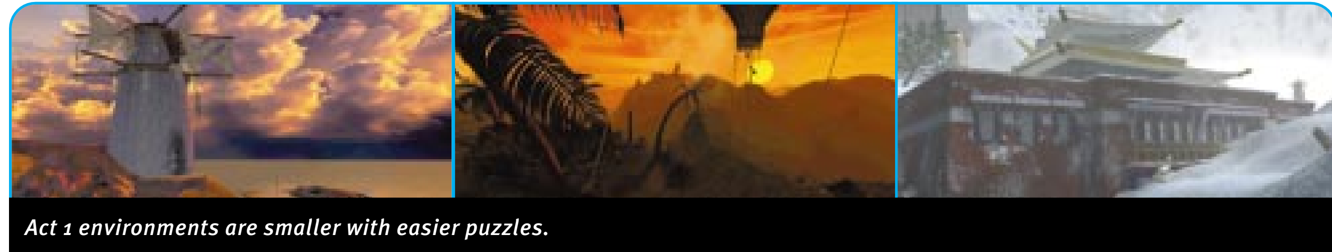
Competitors: See the 3D Engines List at http://cg.cs.tu-berlin.de/~ki/3de_noframes.html for a long list.

Jo e ma P ojec 3: Legac of Time

by Greg Uhler

Upon completion of THE JOURNEYMAN PROJECT 2: BURIED IN TIME, the team at Presto Studios was completely exhausted. Many of us had transitioned directly from developing the original JOURNEYMAN PROJECT into the production of J2. The last thing on people's minds was *another* JOURNEYMAN. Or so we thought...

As we discussed our product strategy for the next few years, it became evident that although a few veterans were burned out on JOURNEYMAN games, many of our



Act 1 environments are smaller with easier puzzles.

employees were anxious to explore new avenues within the JOURNEYMAN universe. While some of the veterans started a new project, the "new blood" in Presto invigorated the remainder of the team, and on January 1, 1996, began development of what was to become THE JOURNEYMAN PROJECT 3: LEGACY OF TIME.

The first step forward in the development of J3 was to take a few steps backward. Before any writing or design could take place, we needed to carefully evaluate the strengths and weaknesses of our previous products. We read customer registration cards, magazine reviews, and fan mail, and we wrote down our own opinions of our games. The result was a list that included many acclaimed features (which we wanted to continue in J3), but more importantly identified strong criticisms that would have to be addressed both creatively and technically (Table 1).

With this list as our guideline, story development and game design began. We envisioned an adventure game in which the purpose of each element was to convey a sense of realism. The player must believe that the world they are exploring is *real*. Navigation would need to be simple, yet extremely powerful. Characters that the player would meet needed to be genuine. Audio required just the right mix of environmental ambiance and music to convey a specific mood. The cinematic cut-scenes would need to be succinct and compelling, with production values on par with traditional films. In the end, we achieved most of our lofty goals, but had we been aware of some of the pitfalls, J3's development could have been easier.

Greg Uhler is one of the founders of Presto Studios Inc., a leading software company located in San Diego, CA. Prior to being the Producer for Legacy of Time, Greg was the lead programmer for the Macintosh versions of The Journeyman Project and Buried in Time. Please visit www.legacyoftime.com for more information.

What Worked?

1. GAME PLAY FLOW. Early in the development of J3, we focused on how players experienced our games. We realized that although we were providing hours upon hours of adventuring, most players were either not feeling a sense of progression during their quest or were confused as to what their immediate goals were. What we needed to do was spread more story information and reward players throughout the game play experience. Our solution was to use compelling, cinematic cut-scenes, something we had aspired to do for quite some time. The challenge of using cinematics was to sustain the suspension of disbelief when we cut from a first-person interactive adventure to a third-person movie.

What we came up with was a three-act structure for the game, both to provide short-term goals and to create a progression of difficulty. In Act 1, players are given an introduction to the main characters and are sent on a specific mission. The early environments that they explore are smaller, the puzzles are easier, and players get a chance to learn the game mechanics (how to navigate, use inventory, and so on) When they complete their mission, a cut-scene reinforces what they've discovered and sets up the critical conflict of the story. Act 2 opens up vast worlds, full of characters (more on that later) and difficult puzzles that are integrated into the environments. As the player accomplishes each of the three main goals of Act 2, a short cut-scene congratulates them, but sustains the emotion of the critical conflict. Act 3 is the climax of the story and closes with several cinematic scenes. Although novice and experienced gamers will progress through the game at different paces, players of any level will feel a strong sense of accomplishment upon completing the game.

2. VIDEO PRODUCTION. For the production of the cinematic scenes in LEGACY OF TIME, we developed two important

TABLE 1. *Past strengths and weaknesses of Presto products.*

Most Acclaimed Features:
- Richness of detail overall
- Immersiveness
- Strong Story
  - Depth
  - Completeness
- Quality of graphics
- Quality of sound/Music
- Graphics/sound *in support of* story
- Arthur concept
  - On-board help
  - Humor
  - Companionship
- Puzzles — logical and challenging.
- Variety of environments

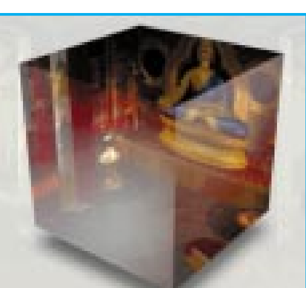
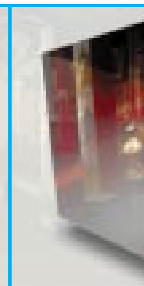
Strongest Criticisms:
- Speed/responsiveness
- Limited/cumbersome navigation
- Need to systematically look everywhere for objects
- Linear game play
- Intimidating interface
- Small view window
- Quality of acting
  - Puzzle difficulty — too hard for beginners/too easy for die-hards

Images from the final scene juxtaposed with the original storyboard.

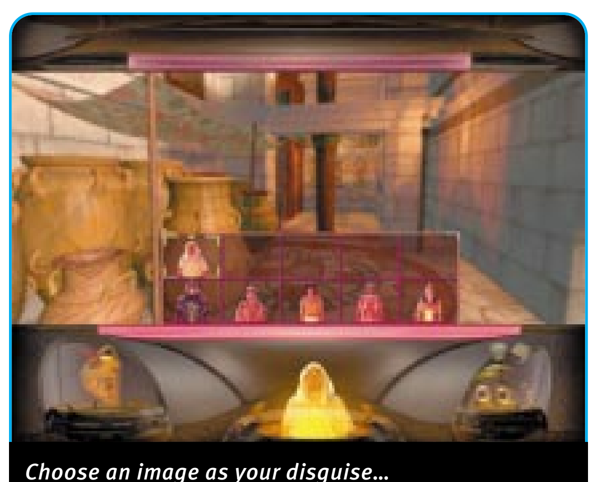
techniques; one involved efficiency while shooting the scenes, while the other was a post-production technique. We knew that the cinematic scenes in the game would require a significant amount of blue-screen video work. Our budget allowed for about two weeks of studio time (stage rental, actors, costumes, makeup, and so on). Our first step was to produce storyboards based on the 30-page script that had been written. Then we totaled the number of shots in these storyboards, knowing that each shot translated into a unique camera and lighting setup. We soon realized that if we followed these storyboards, we would be on stage for almost four weeks. Significantly editing the story was out of the question, so instead we analyzed the storyboards for each of the 13 scenes. We realized that many of the camera angles and lighting setups were similar. For instance, if we had three unique camera angles in a scene (let's call them A, B, and C) the storyboard sequence might be A-B-C-A-B-A-C. To save time, we could shoot the scenes out of order: A-A-A-B-B-C-C. This worked well for us and we completed our shooting in two weeks.

Once all of the video was shot and the final takes had been digitized, we began looking for ways to optimize the compositing and editing of the cinematics. In our previous games, we had composited all of the material that we digitized, then edited it together. For LEGACY OF TIME, our Digital Video Specialist decided to edit the blue-screen footage first and give only the necessary footage to the compositing artists. This was beneficial in a number of ways. First, the amount of material that needed to be stored was significantly less. Second, compositing artists could create their special effects and know that nothing would end up on the cutting room floor. Lastly, with the blue-screen edit as reference, compiling all of the composited shots together literally only took a few hours.

3. NAVIGATION. Early in the development of LEGACY OF TIME, we established two goals for how the navigation needed to work; the method needed to be intuitive, as well as

'Nodes' were built by rendering the six faces of a cube, then adding the bottom, the front, the left, the right, the back, and the top. The final image is the full spherical node.



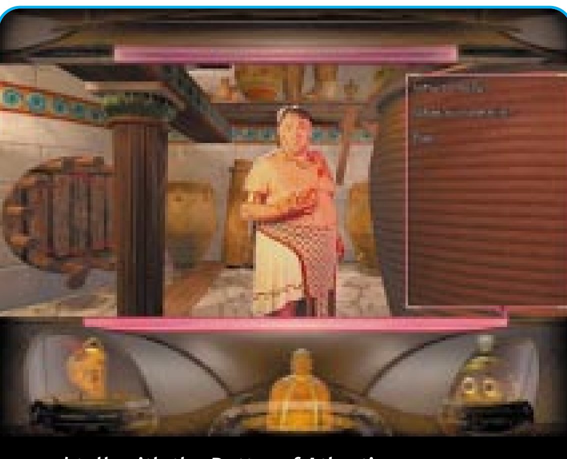
Choose an image as your disguise...

responsive. We felt that a VR-type of technology would fulfill these needs and create a great sense of immersion. Our first step was to take a room from BURIED IN TIME and prototype the new navigation idea using Director and QuickTimeVR. The prototype worked well, but also pointed out some flaws. We learned that VR technologies tend to come in two flavors: cylindrical and spherical. To understand the difference, imagine being inside a can of soup with both ends removed and a label wrapped around the inside of the can. This is what cylindrical technologies are like; they limit your view vertically (so you can't see the holes at the top and bottom) but allow you to look 360 degrees horizontally. Spherical technologies differ in that they put you within a sphere with no holes and let you look in any direction that you choose. As game designers, the spherical method allows us more freedom. We could put objects on the ground in front of the player, create a room with a rope hanging from the ceiling, or design mechanisms above or below the player that need to be manipulated.

Once we decided upon a spherical technology, we needed to connect these 360-degree nodes so that a player could seamlessly walk from point to point. We took screen shots of the view from within a node and tried to match this in Electric Image. The nodes had distortion and correction which was difficult to match in 3D. However, after a few days of experimentation, we were able to match the view from the node. Then we rendered out a walk that connected two of the nodes. We determined that 15 frames-per-second was the optimum rate at which to render the walks; anything less felt jittery, while anything more was overkill. By seamlessly combining the walks with the nodes, we created an extremely intuitive navigation system.

4. BREAKING THE JOURNEYMAN RULES (CHARACTER INTERACTION). One of the tried and true rules in our JOURNEYMAN universe is "Ye shall not alter history." This means that when a player visits a historical location, they may not alter anything, for fear that they might alter the flow of history. What this rule translated to, in terms of game play, was "Ye shall not interact with characters." In J1, we threw the player into jail if they tried to approach any characters in the game. In J2, we gave the player a time-travel suit with a cloaking device, so they could become invisible (as long as they remained stationary.) For LEGACY OF TIME, we wanted to upgrade the suit's technology…

What we came up with was the idea of the Chameleon Jumpsuit. Rather than cloaking yourself so as not to be seen, what if you wore a suit that could display a virtual disguise? This single idea opened up a wealth of possibilities. The player could meet char-

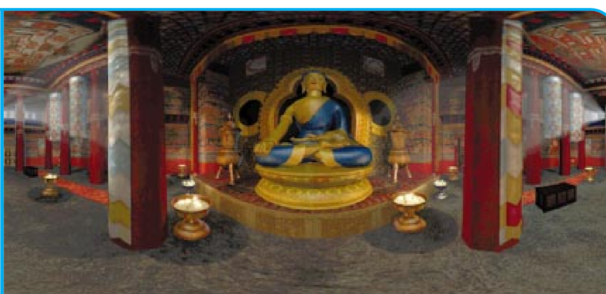


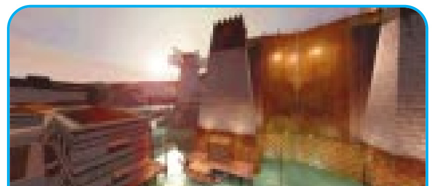
...and talk with the Potter of Atlantis.

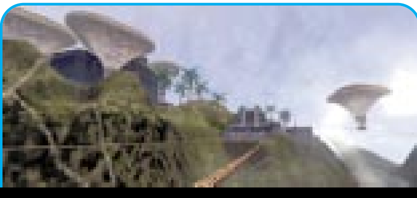
TABLE 2. *Filename conventions*

ER_Filename.EXT

E = Environment (**A**tlantis, **El** Dorado, **S**hangri-La)
R = Room (**D**ocks, **F**arm, **G**reenhouse, and so on)
Filename = a clear, concise description
.EXT = 3 digit extension to define the type of file




The sea-faring city of Atlantis.


El Dorado, the city of Gold.

A Tibetan monastery in Shangri-La.

acters from the past, capture their image and disguise themselves as that character. We could even create a dialog engine so that the player could converse with all of the characters. The possibilities were great, but so was our risk of failure.

Questions arose: Who initiates conversation? Can the player look around while in conversation mode? Does the player have a unique conversation with each disguise that they use? We had to answer all of these questions, but the most important one to answer was, How does the dialog engine work? We didn't want a text parser, nor an extremely limited set of awkward responses. This led us to the idea of dialog topics. For instance, when the player meets the Potter in Atlantis, the Potter says, "Oh, I'm sorry, I'm much too busy for customers today." This greeting brings up the dialog topic, "Why so busy?" As conversation continues, more dialog topics are brought about by the responses of the character. Also, as the player explores in the game, they learn important dialog topics, which they can then use in conversation. In this way, conversation topics grow as a player progresses in the game.

5. Naming conventions and organization. One of the most common problems in our previous productions was organizing all of the digital data that we produced. We needed a method to identify files quickly and easily. So, in the early stages of production, we established the following naming convention shown in Table 2.

Though it may look simple, this naming convention was extremely powerful and efficient. A quick glance at the first two letters instantly identified to which environment and room a file belonged. For instance, if a person was looking for a wood texture that they knew had been used for a boat in Atlantis, they could search for files beginning with AD (Atlantis Docks.) The description would likely contain "wood," while the extension might be .RGB, .LUM, .BUM, or .TRN for the color channel, luminance map, bump map, or transparency map, respectively. As you can see, a lot of information can be stored in the filename itself, saving time both in finding a specific file, as well as not having to open a series of files to find the correct one.

If possible, throw more hardware at it.
–Presto rule of thumb

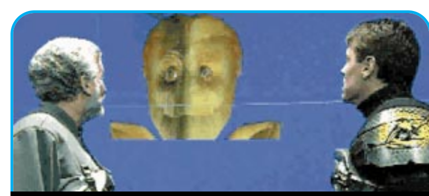
What Didn't Work

1. Expansive outdoor environments. Creating vast outdoor worlds such as Atlantis, El Dorado, and Shangri-La was very exciting from a story and game play standpoint. However, from a 3D graphics production standpoint, outdoor worlds inherently have many difficulties. The first concerns continuity. It had been common practice for our 3D animators to turn objects off, such as the interiors of rooms and distant buildings, in order to decrease render times. However, in these new outdoor environments, nearly all of the models and textures in an environment were visible all of the time. Nothing could be hidden or removed because when the player is outdoors, everything can be seen from our 360-degree nodes. Despite our animators being extremely careful about turning objects on and off, many continuity errors regarding objects, lighting, or shadows had to be fixed by rerendering or rotoscoping.

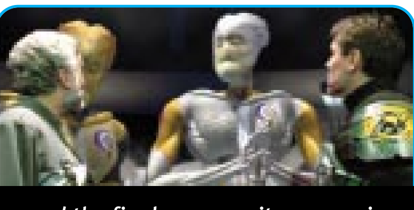
Second, since all of the geometry must be visible, both the software and the machines that are used to render the images must be capable of supporting several million polygons. Luckily, Electric Image Animation System was capable of handling many polygons, so software wasn't an issue. As for hardware, we originally equipped our render farm (unmanned machines that would render images 24 hours a day) with three PowerTowerPro 225 machines, each with 128MB of RAM and 2GB hard drives. As each outdoor environment came together, we realized that the we had severely underestimated our hardware requirements. In each machine, RAM was increased to 256MB, then 384MB, and finally to the machine's physical limit of 512MB. We also added four more PowerTowerPro 225s and 4GB hard drives to split up the rendering tasks. Doubling our hardware also doubled our hardware costs, but it was the only answer if we wanted to finish the product on time.

2. Grouping of personnel. The layout of the office for the production of Legacy of Time was organized by department. We had individual rooms for modeling, texture mapping, and animation. This allowed each department to share production techniques within their own group. Typically, each department had three people. Since there were three time zones, each person in a department had ownership of one of the time zones. This all seemed very logical and straightforward as production began.

About six months into production, however, two problems became apparent. First, there seemed to be a lack of team spirit for each of the time zones. If a problem arose in, say, Atlantis, it was difficult to track down the source of the problem because people tended to defend their department rather than come together as a team and search for a solution. Second, people began to get burned out on what they



An edited cinematic scene before compositing...



...and the final scene as it appears in the game.

were doing. For example, our texture artists had been solely creating and applying textures for six months, although they were quite capable of helping out with the modeling and animation.

It may have worked better if we had created three production teams, each with their own room — one for Atlantis, one for El Dorado and another for Shangri-La. This may have created more of a team spirit for each time zone and allowed for more sharing of the modeling, texturing, and animation duties.

3. A BLUE SUIT? For the production of the Chameleon Jumpsuit used in our video shoot, we hired Don Pennington Inc., an Emmy award-winning special effects company in Burbank, Calif. In our first meeting with Don, we discussed the use of the suit, its appearance, our eight-week schedule before the video shoot, and the fact that we were shooting on blue-screen. He was very professional and had an extremely talented team of artists. Six weeks later, the suit was finished and Don sent photos for us to review. To our horror, the suit had been painted blue. We knew this would never work on the blue-screen. Apparently, we hadn't been clear enough to Don; he had assumed we were shooting on green-screen, as many effects companies do, and remembered "blue" from our first meeting as being the color we requested for the suit. Don was very apologetic, and we came to the conclusion that we were equally to blame for the mix-up. We split the cost to have the suit repainted (with even more care and detail.) The suit was finished in time for our video shoot, and we learned that every detail should be put down on paper to avoid delays and mistakes.

4. AUDIO R&D. One of the mistakes that we made in the production of the game was not researching compression technology for some of the audio in the game. Although our ambient sounds and music are 16-bit stereo, all of our character dialog, footsteps, and sound effects are 8-bit mono. If we had spent more time researching audio technologies, we might have been able to use higher quality audio for those areas of the game. This would have put the finishing touch on the game. I know our next production will definitely include an audio R&D phase.

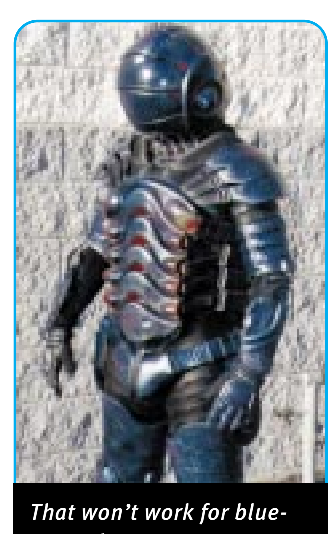
5. MORE PREPRODUCTION PLANNING. We feel the key to successfully developing a product (not to be read as "developing a successful product") is to make as few mistakes as possible during production. This means that the planning stage is the most important phase of production. Our planning stage occurs during story writing and design. All of our designers commented that they felt rushed during LEGACY OF TIME. The result was that there were several difficult issues later in production because of poor planning. Obviously, not every issue can be foreseen, but many would have been minimized by our taking a little more time in preproduction.

What Saved the Product?

Our lead programmer for LEGACY OF TIME was responsible for the cross-platform design and PC implementation of the product. We also hired an experienced programmer for the Macintosh implementation. With nine months to go and only a prototype put together, our lead programmer decided to pursue other interests. We were left in a difficult situation, and our publisher was extremely nervous. Turning to our Macintosh programmer, we asked him if he was interested in taking over the cross-platform design. After careful consideration, he accepted the challenge. We hired a skilled PC programmer to fulfill the PC implementation and were off and running.

Looking back, this was the best thing that could have happened. Our Macintosh programmer was driven, efficient, detail-oriented, full of brilliant ideas, and never said, "It can't be done." Without him, the product's schedule would certainly have slipped. And the game became fast, polished, and feature-packed due to his ingenuity. ■

The Chameleon suit being built...



That won't work for blue-screen!

The final paint job.

by J.C. Herz

High Concept Disease

Like most people who write about digital entertainment, I get a blizzard of press releases every month from game developers. The releases usually go something like this:

First, there is is an exclamatory headline, in ALL CAPS and punctuated by lots of exclamation marks, announcing that this product is the Next Big Thing. The release proceeds with a couple of paragraphs about how the game is a breakthrough in its genre and loads of turbo-charged prose about the technology: game engine, pixel count, graphic virtuosity, and so on, and so on. And this is supposed to get me really excited about the game. But more often, it reminds me of that scene in *Big*, where Tom Hanks, playing a kid trapped inside an adult's body, looks at the prototype of a really complicated, expensive toy and says, "I don't get it. What's fun about that?"

I mean, sure, everyone likes pretty graphics. But at the end of the day, if it's just another DOOM clone, who cares? You can pack more pixels onto the screen, but if there's nothing original in the game play, if the game harbors no independent spirit or innovative design, there's nothing to brag about. It seems as though the authoring tools are getting more and more powerful, and designers are getting more and more lazy. Anyone can talk about advances in modeling, voxels, antialiasing, and all the rest of it. But do those technical statistics make a game any more fun? Have we made any serious breakthroughs in game play since the arcade salad days of the 1980s? Have we made any quantum leaps, fun-wise? Is POSTAL any more thrilling than ROBOTRON? I think not. It's just better looking.

Which is why all of us wax nostalgic for the "classics," I suppose. Because fifteen years ago, graphics pretty much sucked. So you damn well better have had some heart-pounding game play. Because if it wasn't inherently, structurally fun, you were nowhere. The extreme limits of the available technology forced programmers to actually think, to bang their heads against the wall about game design. And look what we got: PAC-MAN, TEMPEST, DEFENDER, ASTEROIDS, GALAGA. Games that arguably stand up to the orgies of texture-mapping and merchandising currently available. Because it was impossible for games of that era to coast on eye candy and a great marketing campaign. The standards for innovative game play, in a very real sense, were higher. Those games were more different from each other than today's. Because apparently, the game business has succumbed to High Concept Disease, transmitted in some nasty backroom encounter with Hollywood ("Yeah, it's like MYST meets… RIDGE RACER. TOMB RAIDER… with an Asian chick. It's like MORTAL KOMBAT… with a twist.") Inevitably, that's what happens when the financial stakes rise to a certain level and the payroll balloons. But it's sad to see an industry this young in a rut this deep.

After all, this is no time to be conservative. At the moment, everyone with a couple of SGI workstations is piling onto an audience that's completely saturated. Adolescent boys only have so much allowance money to spend, and there are dozens of DOOM imitators squeezing them for it. You can't just turn up the attitude — we've hit the ceiling, attitudinally, and it's called DUKE NUKEM. It's time to turn a corner. Given that the market is sclerotically glutted, this industry's long-term survival requires that game developers code their way out of the friggin' box.

And that doesn't mean hauling out a stack of market research that says the female market is underserved or that there's a exploitable niche of retirees with personal computers. This isn't about dragging out the Ouija board to determine what will sell. It's about forgetting the formulas for a second, maybe even turning off your computer, staring out the window (if you have a window), taking a trip, or maybe like, reading a book.

It's a vision thing. Everyone's looking for inspiration in the same places. Look elsewhere. Everyone's taking the same risks. Take some different risks. You'll make mistakes, but they'll be new mistakes. Interesting mistakes. They'll teach you stuff. No one can learn anything new cranking out another "Mech" title. (I await a barrage of venomous e-mail from incensed "Mech" animators — hit me with your best shot.)

There is so much talent out there — odds are that if you're reading this, you're a highly creative person who, instead of engineering database software or going to law school, chose to work on videogames. It was an insane and inspired choice. You have no excuse to be conventional. If you're reading this, you probably fell in love with videogames because they were incredibly thrilling and fun and unlike anything else you'd experienced. So you have no excuse to stamp out cookie-cutter products. If you're reading this, you probably spent a lot of time learning to use highly specialized, arcane technical skills. Why use that hard-won expertise just to showcase new technology? The technology is not of prime importance. It's only important because it allows you to express your vision, which is the real point. It's not about tools.

Which is to say: Ask not what the medium can do for you… ■

J. C. Herz is the author of Joystick Nation: How Videogames Ate Our Quarters, Won Our Hearts, and Rewired Our Minds. *She can be reached via e-mail at joystick@interport.net.*