gd

GAME DEVELOPER MAGAZINE

NOVEMBER 1997

Back

GAME PLAN

A Pox Upon Patents

Like authors, composers, and other creators of intellectual property, software developers derive their living directly as a result of what they put down on "paper." The registration of ideas. And as such, it is the responsibility of software developers to understand the legal basis which protects their work — and which can potentially rob them of their work.

Recently, Dave Perry, president of Shiny Entertainment (creators of EARTHWORM JIM and MDK) found himself the target of possibly unwanted attention. He caused a minor firestorm after a recent interview with Robin Ward was posted on the Video Game Design web site (http://www.videogamedesign.com/design/davep2.htm). In the interview, Dave stated that he intended to patent the engine technology behind Shiny's upcoming MESSIAH title. In his words, "It is actually the first demonstration of perfect real-time 3D tessellation/deformation/skin/interpolation/volumetric lighting running at full speed all at the same time in just software." I say the attention is "possibly" unwanted, because it's conceivable that his primary goal was to stir up some publicity for himself, his company, and MESSIAH. In any case, I'll bite. The situation reminds us that our code is subject to intellectual property laws that many feel work improperly. As such, it's a good time to review the conditions under which software may be patented.

First and foremost, if you have questions about this subject, run down to the nearest bookstore and pick up Stephen Fishman's *Software Development: A Legal Guide* (Nolo Press, 1993). (There's a second edition that's about to come out that I'm definitely going to buy.) This subject is complicated and open to much interpretation, and the more you educate yourself, the better position you'll be in to spar with other developers about the merits (and otherwise) of the system.

In contrast to software copyrights, which protect the expression of an idea, system, or process (but not the idea, system or process itself), patents go much further. Patents keep others from using the idea, system, or process itself, and this is where the confusion begins. Many people compare code to words, and wonder where the government gets off awarding patents for a sequence of instructions when it would never do so for the sequence of words that make up a book, article, or poem. The difference comes from the way the instructions affect hardware and related devices. Because software is used to control hardware, it is a method or process, and therefore subject to patent laws.

Back in the '50s, the PTO routinely rejected patent applications that were based on algorithms, based on the fact that they are immutable — they are "laws of nature." Now however, software patents are issued for techniques based on algorithms when the software uses the algorithm to control hardware in specific, limited ways. In other words, the algorithm may not be patentable, but the application that uses it may be. It's a fine line the PTO walks, but that's the system we have in place today.

Whether Perry has a patentable technology is certainly not clear from his statement above, and it's unwise for anyone to dismiss the technology out of hand without understanding it thoroughly. If Shiny's engine is that different and eligible for a patent, and the company is awarded one, so be it. C'est la vie, that's their right.

However, I for one am no proponent of software patents. I've heard my share of software patent nightmare stories, and I don't think it helps the growth of the industry to spend more time and money on legal fees. The more patents are issued, the more everyone will tiptoe into development, making sure that their software doesn't inadvertently tread on someone's patent.

Let's hope that Perry is not serious about the patent, and simply desires more prerelease attention for MESSIAH. You know the old saying, "there's no such thing as bad publicity...." ■

Alex Dunne

Game Developer — On the Front Line of Game Innovation

EDITOR IN CHIEF	Alex Dunne *adunne@compuserve.com*
MANAGING EDITOR	Tor Berg *tdberg@sirius.com*
EDITOR-AT-LARGE	Chris Hecker *checker@bix.com*
CONTRIBUTING EDITORS	Brian Hook *bwh@wksoftware.com*
	Josh White *josh@vectorg.com*
ART DIRECTOR	Azriel Hayes *ahayes@mfi.com*
ADVISORY BOARD	Hal Barwood
	Noah Falstein
	Susan Lee-Merrow
	Mark Miller
COVER IMAGE	Real 3D
PUBLISHER	KoAnn Vikören
ASSOCIATE PUBLISHER	Cynthia A. Blair (415) 905-2210 *cblair@mfi.com*
REGIONAL SALES MANAGER	Tony Andrade (415) 905-2156 *tandrade@mfi.com*
MARKETING MANAGER	Susan McDonald
MARKETING GRAPHIC DESIGNER	Azriel Hayes
AD. PRODUCTION COORDINATOR	Denise Temple
DIRECTOR OF PRODUCTION	Andrew A. Mickus
VICE PRESIDENT/CIRCULATION	Jerry M. Okabe
GROUP CIRCULATION MANAGER	Mike Poplardo
SUBS. MARKETING 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	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

Wal-Mart: Right or Wrong?

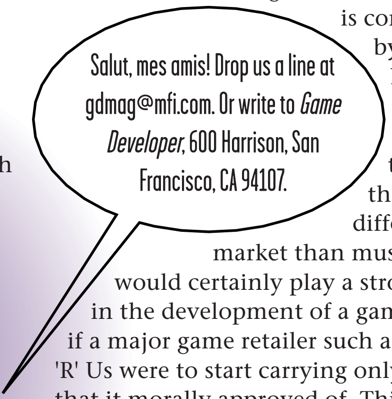
That was a great editorial in the September issue. Wal-Mart has already changed several game titles, both in terms of packaging and in terms of content. Monolith's latest game, BLOOD, had its packaging — as well as the game itself — altered for Wal-Mart (to be less bloody), although the customer could download a patch from Monolith's site to bring back the full gore. CARMAGEDDEN was treated in the same manner. The "gay-pride" Easter Egg in SIM COPTER was also followed by a replacement offer from Maxis.

I recommend subscribing to both *VGA: Video Game Advisor* and *Computer Retail Week*, as both are written by and for retailers and distributors. Both give a very different outlook on the industry from their side. They've been having similar Wal-Mart/censorship articles in their coverage, but with the retail perspective. Frankly, I'm very worried for the industry.

Part of this backlash is caused by the industry's own success. We're finally getting computers and games into the hands of more than 15% of the population, and as a result, into the hands of people who are not all college-educated white males between the ages of 25-35 who make $30-60k a year. While this is a good thing, the current distribution model means that instead of a greater diversity of products, the more we try to get a larger piece of the pie, the more homogeneous our products become, until we all look like Mario — or worse, Hollywood — and the only creative ideas come out of underfunded "art house" game studios. And "internet distribution" is not a magic pill to solve all our problems hitting those niche markets.

I was most pleased to see that for the first time ever in this debate, you didn't call what Wal-Mart does "censorship." Most people in the entertainment industry wave that word like a flag.

John Williamson


Salut, mes amis! Drop us a line at gdmag@mfi.com. Or write to *Game Developer*, 600 Harrison, San Francisco, CA 94107.

Well, I actually hope that the integrity of video games is never sacrificed for shelf space. I don't know what percentage of the video game sales market is controlled by Wal-Mart now, but I think that it's a different market than music. It would certainly play a strong role in the development of a game title if a major game retailer such as Toy's 'R' Us were to start carrying only titles that it morally approved of. This industry isn't only about making a quality game, but also about making money. Once you sacrifice design and quality, the buyer suffers.

We as consumers need to support the little man and deviate from saving a buck at retail giants such as Wal-Mart. One person isn't going to make a difference, of course, but if enough people say, "To hell with them!" then they might get the hint that consumers want more.

David Krueger

I read your article on the "Wal-Mart Effect" and it was right on. Let me tell you about our personal experience with this during the development of SHADOW WARRIOR. It's amazing how many different content restrictions exist now, and not just with Wal-Mart either. It seems that there is always someone new that has to be pleased if you expect to see your product on their shelves, and more are popping up every day.

Before releasing SHADOW WARRIOR, I had to maintain three separate versions of the game. The regular "uncensored" version, the Wal-Mart version, and the UK version — this includes those versions for the shareware release as well!

SHADOW WARRIOR has a parental lock option that allows parents to simply toggle a button to remove all the blood, bad language, and any nudity or partial nudity that is in the game. The parents can also password protect it so that the kids can't turn the option off without the password. This we did pre-preemptively and voluntarily as a standard game feature, and guess what? It still wasn't good enough for some people.

For Wal-Mart, I actually had to hard code the parental lock option to be permanently stuck on and compiled into the final executable in order for the game to be acceptable to them. There isn't even a parental lock menu option in the Wal-Mart version since that is the only version of the game you can purchase there. If you want the uncensored version of the game that you buy at Wal-Mart, you'll have to download a special patch from the 3D Realms web site.

O.K. fine, now we have the Wal-Mart version. Now comes the international market. We came to find out that there is a ban on owning, using, or otherwise depicting shurikens in any media in the UK. I guess people back in the '70s got a little carried away after watching the latest Bruce Lee flick, and this is their answer to the problem. So now, on top of Wal-Mart, we also have the UK version where we actually had to add special code to the game which replaces any and all shurikens in the game with throwing darts at run time. This also required new code and a lot of new art — as if a dart isn't going to hurt just as much as a shuriken. So now, instead of shurikens you'll just have incidents of people nailing each other with darts at every local pub in the UK.

While the executives at Wal-Mart claim that they are not twisting our arms, they actually are, just in a passive way. Everyone knows that in order for your game to be a successful seller, you have to reach the largest market distribution possible. Wal-Mart knows that

▶

Correction

The article, "Implementing Mixed Rendering" in the September 1997 issue of *Game Developer* contained a number of errors. On pages 36 and 38, Figures 2A and 2B have been transposed. Thus, the figure labeled 2A is actually 2B and vice versa. Furthermore, the title for figure 2b (that is, the true figure 2B) is incorrect, as it should have been "Texture mapping composite and double buffer in SW thread."
Not only that, but we omitted co-author Herb Marselas's byline and bio from the article. We deeply regret this omission and any confusion caused by the mislabeling of the aforementioned figures.

they are 10% of our market and that our publishers are going to want that part of the market as much as the other 90% in order to reach as many customers as possible. Well, what's easier? Coercing each game developer and musician separately to produce the content that they want, or simply telling the publishers that they won't carry a product in their 10% of the market unless they develop a product that fits their family values plan? Sure, it's our option to blow off 10%+ of our potential audience. What an option; and surely the Wal-Mart executives know all this.

It may be unfair, but it's not much more than an annoyance if you simply plan far enough ahead. I do expect that this will get worse for each game and music album over time though. We simply viewed this as just another phase of the project before the final release of the game, but it does take up some of your precious development time in both implementation and thorough testing of that version to make sure that there aren't any forgotten booby prizes waiting to spring up and cause you trouble with the Wal-Mart Gestapo.

As Wal-Mart and other chain stores gain a larger percentage of the retail market, I expect the requirements to become progressively worse as their power increases. Who will regulate them on our behalf? And when can we expect retail marketing geeks to arrive on our door steps telling us how we must design our games or write our songs?

Jim Norwood

With regards to your editorial "The Wal-Mart Effect," I believe that you're worrying over nothing. No game developer has a right to be on a store's shelves; remember, the store has to *buy* the game to sell it. If a developer would prefer to change their product to sell more units, that is their prerogative and it illustrates their priorities. You ask, "Who knows what consumers will do if a major retailer stops stocking a certain popular game?" Would the world come to an end if a consumer couldn't buy a copy of DOOM, MYST, or BARBIE FASHION DESIGNER at their local retail store? A chain showing integrity and sticking to its values is a good thing, even if you don't agree with them. Not everyone will do anything for the "almighty dollar." Developers have the same chance to show their integrity and values; they can change their product to make more sales (and those sales are to stores, not consumers) or they can stick with their vision. When Wal-Mart chooses not to sell a popular CD, that's money they're forfeiting for their corporate values. When John Mellencamp changes his lyrics, he is forfeiting his artistic vision for his value$.

John R. Nyquist

I have several thoughts on your editorial in the September 1997 issue of *Game Developer* on the problems of censorship.

First some facts: Censorship is everywhere. It's usually pragmatic. Budgets constrain libraries and librarians get a feel for the needs and wants of their community. Retailers stop carrying items that don't sell and add items that they hear are selling well across the street or in the next town. You will choose which replies are worthy or not of being reprinted in the magazine. Each of us restrict our own choices in many ways.

Here is an amusing and true story. When I was young, I fell in love with the Tarzan books — not the movies, which bore no relationship to the books. I couldn't get them anywhere. They weren't in print. I had discovered the books in my grandmother's home in a back room that had belonged to my uncle when he was a kid.

Then the head librarian at the Kansas City library removed the few Tarzan books the library had from the shelves because "Tarzan is living in sin with Jane in the jungle." (A person who obviously got her literature from the movies and not from books!) Two months later, my mother presented me with all 24 Tarzan books freshly published by Doubleday! Nothing like a little censorship and notoriety to make something popular. I wonder how many people bought the books for a little titillation, only to discover how prim and proper Edgar Rice Burroughs really was!

Yes, there is a big danger of foolish censorship. A marketer may decide to avoid a product because of sensibilities that are at odds with its customers. That is a foolish marketer. He is inviting competition. With the spread of online and catalog sales, the buying public will simply go elsewhere.

On the other hand, the marketer also doesn't want to go against the mores of the community surrounding it, either. The marketer is out to maximize income, and to do nothing that can jeopardize that goal. And I think we should admire marketers who limit their income because of personal or corporate moral or ethical considerations, even if we disagree strongly with their beliefs. These people are willing to lose sales to local, online, or mail-order competitors.

What does that mean for those of us who design and produce products for the benefit or enjoyment of others? Simple. Know your market. If you are designing a risque game, know that the sales will have to be targeted differently than one clearly targeted at children or people with more "traditional" family values. And vice versa.

Remember that only the government is banned from censorship (except pragmatically, when it is the actual purchaser of information, such as with libraries.) Everyone else is free to do as they please. I don't think any of us would want it any other way.

As Alan Toffler first stated, the future has more choices and options, never fewer. Wal-Mart already has lost market share to catalog sales. I think the Internet will eventually be their greatest competitor and not the store down the street. While I don't believe the Internet, or any technology for that matter, will replace the smell and feel of a good book, I suspect there will come a time when purchasing or renting visual or multimedia entertainment (which includes computer games) in a store will be a (fond?) memory.

Al Baker

Public Apology

Speaking of the September editorial, Alex Dunne misidentified the musician who changed the cover of his Christmas album in order to gain Wal-Mart shelf space. In fact, it was Kenny G, not Yanni, who was pictured with a naked infant. We offer our humble apologies to Yanni, whom we have long considered kinda cool, in a strange, mustachioed way.

BIT BL

NEWS FROM THE WORLD

INDUSTRY WATCH

by Alex Dunne

FROM THE ULTIMA ONLINE news front, Electronic Arts announced that during the beta test, the game established the world record for "the largest number of people concurrently playing in the same virtual world" — 2,850 players. What is interesting, is that the size of the game space (189 million square feet, or about 6.7 square miles according to my calculations) would require 38,000 17" monitors to view it all — almost enough to blanket a football field.

IN (OTHER) NEWS FROM THE WORLD OF PATENTS, Aureal Semiconductor received a patent for its head-related transfer function (HRTF) filter technology, which is used in the company's A3D 3D audio technology. The patented technology, developed by the company's Crystal River Engineering subsidiary, covers the compression techniques used to process 3D positioning data. Aureal considers the patent to be a big obstacle to any competitors trying to achieve the same 3D audio at similar price points.

ACTIVISION ANNOUNCED that it will publish the first three titles from Redline Games, a new company started up by James Anhalt and Ronal Millar. Millar was the senior game designer at Blizzard (worked on DIABLO, WARCRAFT II, and the upcoming STARCRAFT), while Anhalt was a consultant to Activision (programmed on projects MECHWARRIOR II and PITFALL: THE MAYAN ADVENTURE). Redline's first title out of the gate will be an RPG/strategy game.

LET'S DO THE TIME WARP. Activision announced that it acquired the rights from Atari/JTS Corporation to develop titles based on the classic games ASTEROIDS and BATTLEZONE. These two games will join PITFALL and ZORK as spruced up and relaunched old favorites under the Activision moniker. It's just a jump to the left…

TOM RYAN, THE CEO OF SCITECH SOFTWARE, testified before the House

MAGELLAN Studio 1.0

LOGITECH has unveiled MAGELLAN Studio 1.0, a smart software/hardware solution for the professional multimedia authoring market. MAGELLAN Studio 1.0 allows interactive motion control of three-dimensional objects in Kinetix's 3D Studio MAX. The product includes a MAGELLAN 3D Controller as well as dedicated software for interfacing with 3D Studio MAX 1.2. At present, support for 3D Studio MAX 2.0 is in the beta stage and will be available upon the release of MAX 2.0. The MAGELLAN 3D Controller (known as the "Space Mouse" in Europe) is an input device that allows users to intuitively and precisely manipulate 3D objects with six degrees of freedom in CAD/CAM and visual simulation applications.

In 1993, Magellan was chosen to control a robot in space during a space shuttle mission for NASA.

Key features of MAGELLAN Studio 1.0 include dynamic control of 3D objects, cameras, or lights simultaneously in up to six axes; "roll-up" menus for axes constraints, recording control, Move/Scale function, and node position; automatic animation recording, providing real-time motion capture, along with features such as slow motion, clipping, and pause recording; four static buttons for sensitivity control and reset of transformation matrix; four dynamic programmable buttons for direct fingertip access to more than 20 edit and display functions in 3D StudioMAX (default and user-specific button setting are available).

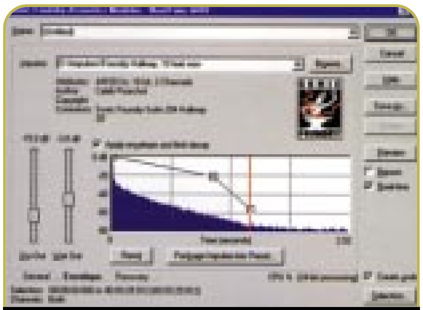
MAGELLAN Studio 1.0 supports Windows NT 3.51, Windows NT 4.0 and Windows 95 and has a suggested retail price of $495.

■ Logitech
Fremont, CA
(510) 795-8500
http://www.logitech.com

Acoustics Modeler

SONIC FOUNDRY has just released its Acoustics Modeler Plug-In, a digital signal processing tool that adds the acoustical coloration of real environments and sound altering devices to existing recordings.

Acoustics Modeler was developed for use with any editor that supports DirectX plug-ins, including Sonic Foundry's Sound Forge 4.0. Rather than simply adding reverb to a sound file to simulate an audio environment, Acoustics Modeler actually incorporates the acoustical responses of a given environment into a sound file. The


The Acoustics Modeler Plug-In.

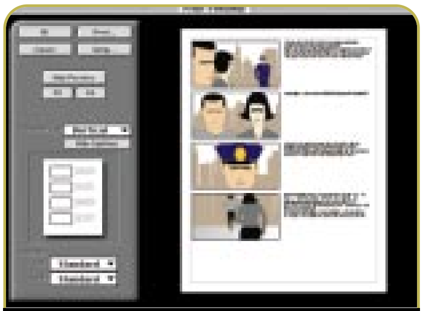
program features a library of dozens of these features, including recording studios, concert halls, warehouses, tunnels, bridges, and woods. In addition, recorded material can be filtered through numerous sound-altering devices such as classic microphones and instrument amplifiers. Users can also record and access their own acoustical environments with the Impulse Recovery Function.

The Acoustic Modeler Plug-In has a suggested retail price of $249.

■ Sonic Foundry
Madison, WI
(608) 256-3133
http://www.sonicfoundry.com

StoryBoard Quick 3.0

POWERPRODUCTION SOFTWARE has released the new version of its storyboarding software, StoryBoard Quick 3.0.


Storyboard Quick 3.0 gives you one, two, four, six, nine, twelve, sixteen frames per page, vertical or horizontal, with or without captions, in four aspect ratios.

StoryBoard Quick features dragable characters, props, and locations, allowing you to previsualize your project. This new version has a number of additional locations, props, and characters. New libraries, new printing formats, and new aspect ratios complement a revised and updated user interface. StoryBoard Quick imports scripts from Final Draft, as well as .GIF, Mac PICT, Windows Metafiles, and bitmap files allowing you to scout locations on the Web.

Storyboard Quick 3.0 is available for Mac or Windows 95. The latest version costs $249, and an upgrade from previous versions costs $69. You can download the demo from the company's web site.

■ PowerProduction Software
Los Gatos, CA
(800) 457-0383
(408) 358-2358
sales@powerproduction.com
http://www.powerproduction.com

New DVD Standard

THE DVD FORUM has proposed the format of a DVD-RAM disc (rewritable DVD), which has a storage capacity of 2.6GB on a single side, to the European Computer Manufacturers' Association (ECMA). The DVD Forum will also propose the DVD-RAM format to other international standardization organizations. The DVD-RAM format version 1.0 was agreed to by the Forum at the end of July, and the Forum started issuing the Format Book in early August. Following the announcement of the format version 0.9 in April, the DVD-RAM working group evaluated the compatibility of the DVD-RAM disc with other DVD disc formats.

The standard states that:

- Playback of DVD-RAM discs on existing DVD-ROM drives is possible simply by modifying the LSIs.
- Data reliability is ensured due to the use of a dedicated cartridge and, for compatibility with DVD-ROM drives, it is also possible to take the disc out of the cartridge.

The DVD Forum consists of Hitachi Ltd., Matsushita Electric Industrial Co. Ltd., Mitsubishi Electric Corp., Victor Company of Japan, Pioneer Electronics Corp., Sony Corp., Toshiba Corp., Philips Electronics N.V., THOMSON Multimedia, and Time Warner Inc.

■ The DVD Digital Domain
http://www.dvddigital.com

Judiciary Subcommittee on Courts and Intellectual Property, speaking on behalf of the WIPO Copyright Treaty Implementation Act. Ryan testified that SciTech, maker of the MGL graphics library, had been threatened by an extortionist who warned that unless the company paid him $20,000, he would post instructions on the Internet explaining how to disable the countdown timer on the trial version of SciTech's Display Doctor. The CTIA would provide copyright owners with more legal means with which to fight unauthorized circumvention of technological protection of copyrighted software, such as the ability to act against web sites that post serial numbers so that pirated software can be installed.

SEGA AND MICROSOFT are working towards an agreement whereby the next generation Sega console system would be based upon Windows CE. Sega's machine will reportedly be based on a 128-bit microprocessor from Hitachi. By basing the console on Windows CE, the companies hope to make it easier for game developers to build titles for Windows-based PCs and the next generation console using a similar code base.

"OVER 1,000,000 SERVED." No, that's not a McDonald's slogan from the '60s; it's the number of unique users Blizzard claims to have logged into Battle.net. The service hit that number just eight months after its launch, and is a testament to the success of DIABLO. According to PC Data's numbers, DIABLO has sold around a million copies to date. Hey, wait, that means every person who bought it…

IT'S AN ODD WORLD OVER AT GT INTERACTIVE, which announced a $10 million global marketing campaign for ODDWORLD: ABE'S ODDYSEE. The announcement is notable in that GTI has already sketched out four sequels to ABE'S ODDYSEE before the first has made it out the door. These games, GTI explains, form a new genre called "Aware Life forms in Virtual Entertainment (ALIVE)". Earth to GTI marketing department: Put down the rubber cement and come out with your hands up. Let's see if the first title is a hit before we get carried away with the sequels.

Texture Transparency and Texture Compression

This month marks the first time that I've had a chance to sit down and write about some various odds and ends that I've been meaning to discuss.

Transparent Textures

A major topic in game development these days is support for transparent textures. Transparent textures have been inconsistently supported by both hardware accelerators and even graphics APIs – Direct3D has a very strong bias for one type of texture transparency (chromakey), and OpenGL supports texture transparency only through alpha testing.

Transparent textures have countless uses, including bitmapped sprites, windows, grates, and any other complex object that is easier to represent with a sprite than actual geometry. Figures 1 and 2 are examples of these, a fan and grate from DESCENT by Parallax Software. The green portions of the bitmaps represent transparency, allowing the background image to show through.

Transparency in textures can be implemented as varying levels of translucency or simply as full opaque/full-transparency "punch outs." The former can only properly be implemented through alpha blending and alpha testing; however, punch outs can be implemented through either chromakey or alpha testing.

CHROMAKEY. Chromakeying is a technique commonly used in the video production world for merging two images. A *key color* in the first image is replaced by the contents of the second image. This concept is the foundation upon which "blue screening" is based. Transparent texture maps implement chromakey by allowing the game to designate the key color. With 8-bit paletted textures, this is trivial since it only requires a single palette entry to be designated as transparent. When rendering the texture map, any texel that is equivalent to the key color is ignored and not rendered. Again, Figures 1 and 2 are examples of this, where the green color represents the key color value, and thus won't be rendered.

Things became more complicated with RGB texture maps and hardware acceleration, since MIP-mapping and bilinear filtering can screw up your chromakey color test. For example, you could designate the key color as 0x808080, but with bilinear filtering enabled, the odds of a texel coming through the filtering stage unmodified is very unlikely. And the simple box filter that most MIP-map generators use will mangle a unique color when found in smaller MIP-map levels.

For these reasons, bilinear filtering and MIP-mapping are typically mutually exclusive with chromakey. Some hardware architectures have added support for a key color range instead of just a single key color, but this isn't a particularly elegant solution since it's prone to accidentally removing other texels as the result of blending and MIP mapping. In some extreme cases, it won't even correctly handle a transparent texel that ends up being filtered to an out of range value.

Ever since RAM costs plummeted lower than Ray Liotta's career, texture compression hasn't been as high a priority for most 3D accelerator manufacturers.

Alpha testing and alpha blending are often more appropriate than using chromakey when implementing textures with transparency.

ALPHA BLENDING AND ALPHA TESTING. I've already discussed the theory of alpha blending in previous columns "Multipass Rendering and the Magic of Alpha Blending," pp. 12-19, *Game Developer*, August 1997), so this month I'm going to show you how to use blending and alpha testing for antialiased transparent textures. Instead of designating a single color as transparent, alpha testing allows you to specify certain bits to represent transparency. A texture map with more bits dedicated to alpha will have the ability to represent increasing levels of translucency. The two most common formats for alpha textures are 4444 ARGB, and 1555 ARGB. The former sacrifices color depth in order to allow sixteen levels of transparency. The latter

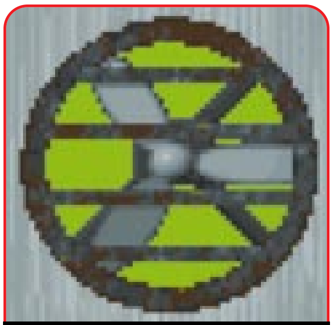


FIGURE 1. *Chromakey textured fan from Parallax's* DESCENT.

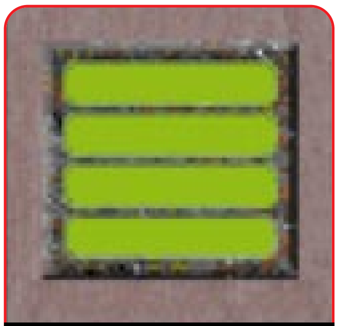


FIGURE 2. *Chromakey textured grate from Parallax's* DESCENT.

maintains higher color resolution, but only has the ability to indicate absolute transparency and opacity.

ALPHA TESTS AND OPENGL. With OpenGL, you enable and configure alpha testing with

```
glEnable( GL_ALPHA_TEST );
glAlphaFunc( GL_GREATER, 0.0F );
```

The alpha function defines what type of relationship must exist between incoming alpha and the alpha reference value to render an incoming fragment. For the preceding bit of OpenGL code, all fragments with an alpha greater than zero will be rendered. Such a function would be used with 1555 ARGB textures, where a texel alpha bit of 1 indicates opacity and 0 indicates transparency. If bilinear blending is enabled, then you can achieve a reasonable form of antialiasing, however, bilinear blending can have odd effects with alpha testing, so you need to be very diligent when rendering textures with alpha blending, bilinear blending, and alpha testing enabled.

ALPHA TESTING AND THE Z-BUFFER. Another caveat is that alpha testing and alpha blending don't necessarily coexist very well with Z-buffering. Take, for example, a case where you have a 4444 ARGB texture. Assume the texture has gradually increasing amounts of transparency as you get closer to the center of the texture, with full transparency at the center (as in, a cutout) and full opacity at the edges.

The intuitive way to render this would be with standard alpha blending.

```
glEnable( GL_BLEND );
glBlendFunc( GL_SRC_ALPHA,
             GL_ONE_MINUS_SRC_ALPHA );
```

The problem occurs when you try to write values to the Z-buffer conditionally based on alpha. The fully opaque parts of the texture need to have depth values written. However, any transparent portions must *not* have depth values written to the Z-buffer. Alpha testing doesn't solve this problem adequately, since a pixel that fails alpha test won't be rendered at all, when in fact you still may want a partially translucent fragment to be rendered but with depth buffer updates inhibited. Unfortunately a mechanism that describes this doesn't exist in Direct3D, nor in OpenGL.

For this reason, textures with varying levels of translucency may be prone to visual anomalies, and this is definitely something that you need to be aware of when working with textures containing alpha.

SOURCE OF THE ALPHA. This is a quick note – be aware of your texture environment and incoming iterated alpha value when working with textures containing alpha. The source of your fragment alpha may not be particularly intuitive and will depend on your texture blend/environment, your texture map format, and the value of iterated alpha. With OpenGL, this is clearly defined. But with Direct3D, however, you'll probably have to experiment with different drivers, because no clear specification exists on this topic.

ANTIALIASING WITH ALPHA. A cool side effect of using alpha bits to represent transparency is that you can get antialiasing effectively for free. Even with a 1555 ARGB texture, you can get antialiasing; bilinear filtering will blend the binary alpha values into intermediates values, depending on texel coverage. This means that textures with "punch outs" won't necessarily have harsh edge aliasing artifacts along the borders between opaque and transparent regions.

Compressed Textures

Texture compression was a really hot topic a couple years ago when RAM prices were phenomenally high. Ever since RAM costs plummeted lower than Ray Liotta's career, texture compression hasn't been as high a priority for most 3D accelerator manufacturers. Some work is still going on in this area; however, I fear that some hardware manufacturers might be going in the wrong direction on this one.

Textures are expected to be compressed either by the game developer as part of a preprocessing step or by the hardware and/or its driver during run time. Both of these methods have some pretty significant drawbacks.

Precomputed texture compression has some obvious benefits — slow but high-quality compression algorithms can be used, and there's very little run-time overhead for using compressed textures. But this assumes that you know what your texture data will look like before your program is even started, and it also assumes that you're using some type of industry standard texture compression algorithm with available tools. Aside from palettized textures, a standard for precompressed texture data doesn't exist. In all likelihood, any compressed texture solutions will be proprietary (such as the 3Dfx Voodoo's proprietary 8-bit compressed texture format).

QUAKE2 needs to dynamically compute texture data through surface caching (in the software-rendering subsystem) and dynamic light map generation (in the OpenGL-rendering subsystem), so the usefulness of precomputed compressed texture data will be limited to games that don't modify their textures on the fly. Also, QUAKE2 is designed to be portable across many platforms, and supporting many different proprietary compressed texture formats isn't particularly desirable or even feasible.

Run-time texture compression, on the other hand, allows the incoming texture to be a lot more dynamic in nature, since the compression doesn't occur until the texture is uploaded to the hardware. The downside to this approach is that there will likely be a significant and noticeable hit on performance every time a texture is compressed and downloaded, discouraging the use of lots of dynamically changing textures. Once again, QUAKE2 relies heavily on texture downloads for effects such as dynamically updated light maps.

Neither of these solutions is perfect, and both have significant pros and cons. Until hardware accelerators that support texture compression hit the market, we won't really know which direction things are heading. So until then, we won't know how to compensate for the various idiosyncratic texture compression implementations that will be heaped upon us.

This month's column has covered a lot of the odds and ends that I've been meaning to address over the past few months. I'd like to thank Matt Toschlog and Parallax Software for allowing me to use their bitmaps. For my January 1998 column, I plan on talking about my experiences with QUAKE2 and the development methodology that id uses. ■

Brian Hook recently had a nightmare in which real-time 3D graphics programming was totally irrelevant to life in general. Fortunately, when he woke up he was still coding QUAKE2 at id Software. Send hot milk and soothing reassurances to bwh@wksoftware.com.

Birthing Low-Polygon Characters

Yeah, I know. My own patience for metaphors in computing is quite short, too; but you have to admit that "birthing" for character creation makes sense. Character design is a control-freak parents' ultimate dream; there are no vagaries of chance here. We artists build a human character exactly how it's supposed to be. Height, weight, sex, clothing style — they're all at our fingertips. Scary, huh?

Let's walk through the birthing process of a human character. We'll do this in two parts: design and construction. This month, we'll start with real-time 3D character design, working toward a solid pencil sketch and a basic construction plan, including face and texture budgets. In January, we'll explore how to actually build a 500-face character based on this design.

Creating compelling, interesting characters is a well-documented topic, so go scam some knowledge from experts.

Character Design

Obviously, character design is a really broad topic, and there are endless approaches to this topic. In fact, for someone starting out, therein lies the problem: too many options. With that in mind, I'm laying out some specific techniques from the sea of possibilities — but if you do it differently, don't feel neglected.

Definition

For those of you who have avoided the term so far, an "avatar" is a representation of a player in any computer-simulated environment. They're often low-polygon 3D models of people in multi-user 3D worlds, but the term also applies to 2D-only "sprite" images of the player. The term isn't usually applied to games, but if it were, the player's character is an avatar. For example, Mario is an avatar.

WHERE TO START? The blank-page block can be very strong when an artist is facing a character design job. Not only are the starting options overwhelming, but character design is more than visual, so we'll have to do some nongraphic creation.

For example, look at Lara Croft from TOMB RAIDER. For a full character design, we'd want to know much more about her than her appearance: Her walking style, her interests, her natural abilities, what kind of people she likes, her accent, her choice of words... all kinds of nonvisual detail should be (and probably was) defined.

We design characters in two steps.

- First, we collect all kinds of detail, even if it's not directly relevant to drawing the character.
- Second, we develop graphic representations: pencil sketches, 3D models with textures, and animations.

A DAB OF THEORY. There's a whole lot of theory that can be invoked for character design. Here's a tiny taste (see the sidebar, "Homework: Character Design Research," for more sources).

The first lesson for artists-turned-designers is, perfection is boring. Think about any cool character you know. Most of their interest comes from flaws and weird quirks. These flaws can be in many forms: physical, mental, emotional, societal, and so on.

For example, take a look at the comic book character Archie, the blandest, most basic good character we can imagine. On one hand, he seems plain and basic; but actually, he's far from perfect. He has a crappy old car, is ridiculously indecisive about girls, and seems to have serious issues with school (Mrs. Grundy). These imperfections make him "real," and even normal.

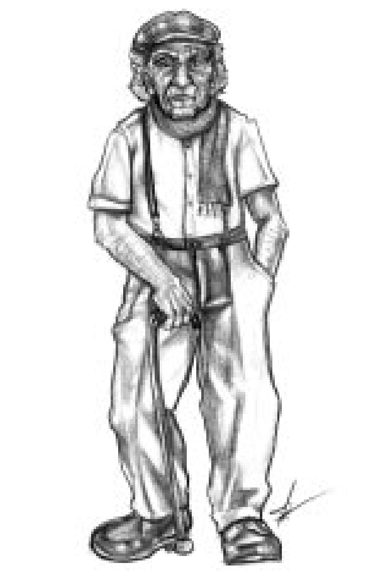


FIGURE 1. *Pencil sketch character.*

But Archie is nothing compared to the twisted psyche of his best friend. Jughead's strange, ugly appearance, suppressed or retarded sexuality, freakish obsession with eating, and all-around spaciness are exaggerated flaws that real people possess — and they make for all kinds of rich storytelling.

FINDING CHARACTERS. So where do you find a library of character traits? Everywhere. People-watching, with a critical eye toward unique personalities, is a great way to get a whole pile of interesting, fresh elements for character design. Spend an hour at an outdoor café with a notebook and see what you can find. Here are some suggestions to keep the experiences fresh:

- Write, don't draw. Instead of desperately trying to sketch out characters as they whiz by, try jotting down one unique detail about each interesting person: "fringe leather bracelet," "dyed-black hair, crispy-looking," "stumpy militant gait," "greasy icky overalls," "thrashed novel, bent-back cover." Later, you can use these reminders to sketch composite characters.

- Pick a theme. Maybe have a "shoe day"; only look at the shoes and try to identify what shoes reveal about a character. Other themes could include hair, accessories (books, glasses, purses), clothes, and motions (walking gaits or unique movements).
- If possible, save images. Obviously, it's handy to have a still reference from which to work, but pointing cameras at random strangers is mighty awkward (and not always appreciated). One quick and easy way to do this is get a camcorder (only $350 these days!) and rest it on the table, aimed at a public space. When someone interesting walks by, record a little snippet. You can then peek in the viewfinder and sketch right there or review it later.

Of course, first-person observation isn't the only source for character design.

COMMUNICATING YOUR DESIGN. So you know what character you want to build; now you have to show other people. Once you have an idea for a character design, it's helpful to get it out where people can see it. But when you're designing someone else's character, you have to know exactly what they want or they're going to be disappointed.

At a minimum, a character design needs to cover the basics. Human or non-human (such as an alien)? He or she? What size? What culture? What style within that culture? Of course, we also need lots of physical appearance details. What skin color? What clothes? (What style of pants? Faded denim or new? Flared leg? And so on.)

Obviously, the head (especially the face) is important for character personality. For example, in the comic strip *Peanuts*, compare Lucy to Sally. The visual differences are all in the face and hair; the bodies are practically identical.

Facial detail is really noticeable when it's missing — witness the spooky lack of facial detail on the Power Rangers' suits. Is it just me, or are those characters really bland because of the missing face? Another proof by negation can be seen in mannequins. They're designed to have no character, and most don't have any facial detail at all. Coincidence? I don't think so.

That's all well and fine, but what if we're building a character that someone else has designed? In that case, our next task is to communicate — to learn what design we're supposed to build.

VERBAL SKETCH. Most people can express a basic character design in a "verbal sketch." This is often the point at which we artists first encounter the character.

For example, I asked my flaky character designer, Jake, to tell me about any old character he could imagine: "OK, um, there's this old guy, Jim, see, and he hates dogs 'cause his wife was killed by a big black Doberman back in '82, and um, his pants are always creased, and he wears this old golfing hat all the time and ummm... suspenders. Yeah, and his shirts are always... they're really clean, like he's super-careful about being clean. He's worried people will think he's a bum because he spends a lot of time at the park, looking for people to talk to 'cause his family all moved to Detroit five years ago, but he didn't want to leave New York."

Verbal sketches are interesting and help give a sense of the character, but do we know enough to build Jim? Not really; we need to know more, so we ask some questions (Table 1).

These questions help a lot. The only answer that we didn't really get was the style — not surprising since poor Jake is being put on the spot. Still, we've revealed that his design hasn't been fully developed yet. That's O.K.; with his specific breakdown of clothes in the verbal sketch, we have enough to work from here.

TABLE 1. *Artist/character designer interaction.*

Artist asks:	*Designer responds:*
How old is Jim?	"He's, uh, 72."
What is Jim's culture?	"He's a Jewish guy who came to the U.S. from Poland when he was seven."
What's his style?	"Uhhmm... he's, uh, normal old guy... not crazy... hell, I don't know!"
What color is his skin?	"White, Polish features; his bushy hair is grey and he's all wrinkly."
What's his build like?	"Thin, but healthy, 5'6" and 120 lbs. He used to be good in track."
How high is the waist of his pants?	"Pretty high, in that old man way."
How long is his hair?	"About 3 inches, curly."
Does he have any props?	"He has a cane, but no glasses."

Sketch. Next, we sketch the character on paper. The sketch forces us to define the character's race, culture, age, style, size, and a lot of other details that the verbal sketch doesn't cover. It's a very important point of reference for teamwork — it's used as a goal for the 3D modeling tasks.

Figure 1 shows a sketch of our character Jim. Now we have all kinds of detail, and the issue quickly changes from the vague, murky, "What does Jim look like?" to the daunting, but attackable, "How do we get all the detail of this sketch into a low-polygon 3D model?" The answer: Understand our limitations, then revise the design to work with what we have.

Gather info (and make decisions). We're now at the rubber-meets-road spot, so we need to find out what the road is. This is usually done through an awkward, difficult series of communications that I outlined in the August 1997 *Game Developer* article, "The Artist Synapse."

At the end of that communication process, we'll know what our game engine can and can't do (or at least we'll know what decisions need to be made). Here's an example scenario:

Our communications efforts pay off, and we now know that the graphics engine is basically a QUAKE clone — it renders a first-person view of a texture-mapped 3D environment with no dynamic lighting (lights are built into the textures), and it can do per-vertex "skinned" animation for character animation as well as normal hierarchy-based animation.

We also encountered our budgets: We have 500 faces and 100k of texture memory to spend on Jim. That doesn't

Homework: Character Design Research

OK, let's say we admit that we need help with this character design stuff. Well, hit the books! Creating compelling, interesting characters is a well-documented topic, so go scam some knowledge from experts. You can find books about character design in any major bookstore or library.

Historical: Character design is one of the oldest and most prevalent themes in art. Storytelling, painting, sculpture, and theater all depend on rich characters. These traditional media have a lot to offer an interactive entertainment character designer. One good way to get a nicely packaged summary of this information is by taking a class from a local school — you might find a class that is devoted to the idea of characters throughout art history.

Film: For game characters, start with character design techniques used in commercial film industry. Just watch some "character study" movies and try to identify what, exactly, makes the characters cool and unique. Of course, you can also find books, articles, and lectures that are aimed at developing characters for films. For example, the director Frederico Fellini specializes in bizarre faces; ask any film buff and they'll point you to an endless stream of relevant video evenings.

Novels: Fiction contains arguably the most deep character design practiced today, so I'd recommend reading about how novelists do it as well. Again, you can just read some excellent books that focus on character development (as most do) and pay attention to what makes the character unique, or you can seek out resources designed for authors. For example, browsing at Borders Bookstore turns up entire books devoted to crime details for authors. These books are shelved in "writing"; they're written to help authors craft realistic behavior for desperate criminals.

Game designers: And, of course, game designers know this stuff. Check out conference notes from lectures on game design, articles for designers, and the like. Better yet, if you know any first-rate game designers in the industry, play newbie and ask them how to do it in person.

Again, character design is a huge area; don't be disappointed if your first design attempts aren't pro quality. Don't forget that there are lots of resources to help you, including professional game designers; they often have a really good grasp of character design.

Whose Job Is This, Anyway?

Some will argue that character design isn't really the artist's job — that's what game designers are for, they say — but it's frequently assigned to artists anyway. And, yes, you have to admit it's just plain fun, so we fun-loving artists naturally want to be involved in it.

How do artists end up designing? In large projects, ignorant producers and publishers tend to assume that any professional artist could whip out a really cool character from blank screen, especially for demos, "B" games, and other non-mainstream projects.

In smaller projects, the job roles aren't usually so strict, so artists do some design, designers can create some art, and together they cover all the bases.

sound like much texture memory, but we also learned that Jim isn't the lead character, so 100k is enough texture memory for such a secondary character. Also, we found out that we can use 8-bit textures, each with its own palette; that means a 256×128 texture only uses 32k of memory.Here's some more important decisions that were made for Jim:

- ***Range of motion:*** A talk with the game designer tells us that Jim doesn't need any complex animation. He's going to limp around and wave his arms a bit, but won't do anything that requires too much detail in the hands or head. This means that we don't have to design the character to be capable of cartwheels; this detail simplifies the joint design, but it also affects texture mapping. Why texturing? For example, if Jim were to bend deeply at the knee, we would have to design textures on the knee that looked good at full bend *and* at normal straight-leg positions. Knowing that he's only going to bend 90 degrees (for sitting), we will create textures that look good from those angles.
- ***Speech:*** The difficult decision about facial animation was made early on — we're using cartoon bubbles for dialog, so we won't animate speech. This will save lots of hard work doing lip-synch (obviously at the expense of realism).
- ***Skinned joints:*** Even though we could do "skinned" vertex animation, we'll be using intersecting joints. "This character is a minor role, and its movement is relatively small, so intersecting joints probably won't look too bad," our art director reassures us. After some questioning, we find out that it's really a performance tradeoff: Animating all 500 vertices can easily mean a frame rate penalty once the math gets to be a heavy load. For the lead characters, it's worth the performance hit, but not for a minor character like Jim.

FINAL DESIGN SKETCH. Based on the information we've just gathered, we can now sketch a more do-able version of Jim. We'll strip out some of the detail that isn't very achievable — for example, the frizzy hair on the sides of the head would have been difficult represent satisfactorily, and it's not critical to conveying Jim's personality, so we could just change the design rather than fight the limit. For other problems, we may choose to fight for a certain important detail, but in this case, it's probably not worth it.

FINAL APPROVAL. Next, we show our final sketch to the art director for approval. As usual, the director is wildly excited by its innate beauty and our rare genius implied, so the sketch is approved instantly, and we get our usual huge bonus, utter worship, and tearful hugs aplenty.

O.K., perhaps that's a tad unrealistic. More likely, the art director expresses some appreciation, but also wants some changes. After these revisions, eventually, you'll get a stamp of approval.

Once the approval festivities are over, our final step is organization. We gather the sketch and the face and texture budgets into one spot — a folder in the appropriate directory, or a single sheet of paper — so we can quickly and easily find them. This folder is the symbolic end of the design phase — we've completed Step One. From here, we get into production and take this pencil sketch to 3D. That's covered in January's column.

Modeling: Basics

Where to start? Let's start with an overall approach. In practice, we'll usually start with an existing real-time 3D human model and edit it to match our sketch. Depending on how similar our characters are (and how carefully we can edit the geometry without disturbing mapping), we don't have to remap or rebuild large parts of the model. But don't worry, I'm not going to pull that kind of shortcut here — we have to start from ground zero.

So we're going to build our character from scratch, aiming for our face-count budget for each part: head, torso, arm, and leg. For texturing and mapping, we're using only Adobe Photoshop and 3D Studio MAX, so we can't do any 3D painting. We'll just use standard projection mapping techniques.

We're going to use cylindrical texture mapping for the head, because we really want the ears and sides of the face to have full detail, but we'll do planar projection mapping for the other parts. This is a trade-off, because planar projection will lose detail on parts of the body. Still, we've decided that the loss of detail on the sides of the arms and legs is tolerable. (Besides, painting planar-projected textures is a lot faster and less error-prone than cylindrical texture mapping).

Let's Divide Our Budgets!

Oh boy! Now we get to divide up our allotted budgets: 500 faces and 100k of texture memory. Let's start with face count.

In general, the head and torso are more able to "flex" their face counts, and since people are generally looking at the upper body and head most, we'll use a disproportionately large number of faces in this area. By contrast, the arms and legs will be modeled as simply as possible, while still implying plenty of detail. Another obvious reason we want to keep arms and legs simple is that there are two of each of these. Any face we add to the design of one arm means double faces in the final model. Let's divide the body into

a few parts and assign face counts:
- Head - 148 (includes neck stump)
- Torso - 102 (chest, pelvis, stumps for joints)
- Arm - 66 (includes hand, elbow joint, shoulder stump)
- Leg - 68 (includes foot, knee joint, hip stump)

So we're using 250 faces for the head and torso, plus two arms (132) and two legs (136) equals 518 faces. That's close enough to our budget for a first pass.

What kind of arm can we model with 66 faces? Nothing fancy — it'll have a closed-fist or mitten-style hand, and no real wrist joint modeled. Same for the other geometry — basic modeling will have to do.

For texture memory, let's start by deciding what we'd like. If our character is only 100-pixels tall, we don't really need very large textures. With 100k, we can have an ample 256×128 cylindrical map texture for the face, leaving 64k for the body. Let's assume left and right arms and legs are identical, so we'll use a pair of 64×128 textures for the front and back of the arm, and another pair for the leg, and a pair of 128×128 textures for the front and back of the torso. Table 2 shows the numbers. We're using seven textures that add to 96k of memory.

TABLE 2. *Budget numbers for Jim, the character from Poland.*

Number	Size	RAM	Where used
1	256x128	32k	Head (cylinder map)
2	64x128	16k	Arm (front and back)
2	64x128	16k	Leg (front and back)
2	128x128	32k	Torso (front and back)

The End.... Not.

We've covered a lot of ground this month. Out of thin air, we extracted a do-able real-time 3D character design. We defined our basic approach — decided on our mapping methods, broke the model down on a per-part basis, picked important areas, and divided up our budgets accordingly.

In January's column, our task is clear: Build a textured 3D model from our sketch and within our budgets. Here's vaguely how we'll do it:
- We'll create the textures and geometry for each part.
- We'll assemble the parts into a hierarchy, adjust joints, test a few animations, and test in the 3D environment.
- Finally, do revisions until the model is approved. ■

Josh White runs Vector Graphics, a real-time 3D art production company. He wrote Designing 3D Graphics *(Wiley Computer Publishing, 1996), he has spoken at the CGDC and cofounded the CGA, an open association of computer game artists. You can reach him at josh@vectorg.com.*

Cop a Feel.... i h

hap.tic (adj.)(1890)
{⇠haptesthai ⇢ to touch}
1: relating to or based on the sense of touch
2: characterized by a predilection for the sense of touch ⇠a haptic person⇢
3: to be out of haptic with reality

Article by Chuck Walters

Hap ic Pe iphe al !!!

Seven peripheral manufacturers are prototyping and/or shipping more than nine force-feedback game devices for home use, at or under $200. Three companies are supplying the force-feedback technology: CyberNet, Immersion, and Microsoft. With force feedback ramping up and three implementations available, Direct X has stepped in to standardize the API used to control these devices. This article will give you a general overview of the tools and techniques used to implement force feedback in your games. It will focus on generic design issues and DirectX 5 implementation. Before diving into the nitty-gritty code work, however, we will review the hardware.

Two force-feedback joysticks and three force-feedback wheels are expected to be in retail outlets by Christmas 1997. At least six more force-feedback devices are planned for next year. CyberNet, Immersion, and Exos (acquired by Microsoft) have been offering products for high-end simulations and research for years, but CH Products was the first consumer product to make it to the retail market. The companies in Table 1 have at least a working prototype slated for home PC use.

Some of the companies listed in Table 1 also distribute force-feedback devices for game consoles and arcade machines. Companies such as Happ Controls (the largest manufacturer of arcade force-feedback devices) use technologies similar to those being discussed, but they won't be the focus of this article. For a more complete view of the market, take some time to browse the research sites listed at the end of this article. The Intel site has the specifications for the Open Arcade Architecture Coin-op Reference Platform also known as the Arcade PC standard (which includes force feedback).

Mechanics

Not all force feedback devices are created equal. The more expensive devices are able to produce high-frequency sensations at a cost of a few thousand dollars, compared to a few hundred for the home versions. A hefty portion of the cost difference is spent reducing the compliance (play) between the actuators (motors) and the stick. The following sections discuss the interface between the actuators and the stick.

Components. The play and durability of a force-feedback device depends, in part, on the material and tolerance of the machining. Cheaper materials, such as plastic, are pliable, do not hold up to high-tolerance machining, and wear faster. Here, the material limits the performance. More expensive alternatives, such as composites, aluminum, and alloys, can hold higher-tolerance machining. Still, just because quality material is present doesn't mean that it has undergone the costly high-tolerance machining. Quality material has the added benefit of holding up better

TABLE 1. *Availability of force feedback devices.*

MFG	FFB Engine	Product Name	Interface	SRP	ETA
Act Labs	Immersion	Racing System	GP	$99	9/97
CH Products	Immersion	Force FX	SP & GP	$150	Retail
		Racing Wheel FX	SP & GP	$150	11/97
Interactive I/O	Immersion	Virtual Vehicle Tdi	SP or USB	$995	Direct
Logitech Inc.	Immersion	WingMan Force	SP or USB	?	?/98
SC&T International	Immersion	Ultimate PER4MER	SP & GP	$170	11/97
Thrust Master	Immersion	MotorSports GT	?	$200	11/97
Cybernet	Cybernet	Real Feel Wheel	SP & GP	$200	Direct
		Real Feel Yolk	SP & GP	$200	Direct
Microsoft	Microsoft	SideWinder FFB Pro	GP	$150	9/97

Interface Legend: SP = Serial Port GP = Game Port USB = Universal Serial Bus

to general wear and tear. Let's open a force-feedback input device and examine the inner workings.

Gimbal. The gimbal connects the stick to the transmission. Force-feedback joysticks generally have more play than steering wheels because of their dual-axis motion. The biggest contributor to gimbal play are the slots of a slotted bale. Double-slotted bales (Figure 1) are the loosest. Immersion has patented a slotless gimbal, shown in Figure 2. Keep an eye out for this design in retail sticks over the next year. Currently, it's only available in research-quality devices.

Transmission. Actuators are connected to the gimbal by the transmission. Geared transmissions must have play or the gears will bind. Tight gears require precise machining. Therefore, cost becomes reliant on the material and machining commodities. A cable/belt drive is a higher-quality method of transmission. Since there are no gears that can bind, this method has the potential for a very tight force response and much less noise (gears are quite noisy). The counter rotational requirements of force-feedback transmissions are demanding, so either the cable/belt will need to be extremely strong or devices will need easily accessible adjustment screws for keeping the cable/belt tight.

Actuators. Force-feedback joysticks use two motors, which are similar to those found in fax machines and printers. They exert about 1 lb. of sustainable force per motor, peaking at around 1.5 lbs. Force-feedback wheels have one motor that can sustain 3-4 lbs., peaking at around 5 lbs. Cheaper motors normally exhibit higher friction, so they dampen out subtle forces, causing poor or unperceivable response.

Circuitry

Device manufacturers are scrambling to produce high-quality feel at a low cost. Currently, circuitry improvements are on the back burner because they don't satisfy the manufacturers' immediate goals — what they have works well. When the competition heats up, however, circuitry will be upgraded in the following areas.

Processor. Most force-feedback devices have a microcontroller to control the motors and handle effect computations. Motor control is cut and dry, whereas effect-handling can be improved in many ways. Future processors will handle new types of effects and be able to play more of them simultaneously. There will also be improvements in dynamic effect transitions.

Memory. The amount of RAM determines the number of effects that can reside on the device. Different effects require varying amounts of RAM, so tying the number of one to the other isn't well-defined. To illustrate this point, current devices have on-board memory ranging from a few hundred bytes to a few thousand bytes. A single effect takes anywhere from several bytes to several hundred bytes. Therefore, there is RAM for potentially hundreds of effects, but many devices partition this space to simplify the microcode, thus artificially reducing the total count. There are at least two benefits of increasing the on-board RAM:

- More RAM allows more effects to be stored locally, reducing playback latency.
- More resident effects means that effects can be played simultaneously (added together), creating new effects.

SENSORS. These are the stick position tracking mechanisms. Many devices use potentiometers to track the stick or wheel's position, which is the main reason devices need calibration. Potentiometers are variable resistors that lose resistance as they age, which causes a positional drift in the joystick. Microsoft's Sidewinder uses an innovative digital optical tracking system that is self-calibrating. Digital optical tracking is also faster at position acquisition then potentiometers, which means less time is spent polling the joystick, thereby freeing CPU cycles for other game needs. In addition, fast acquisition time is important for condition effects (explained later), which are dependant upon positional data. Effects such as those that simulate inertia will perform more smoothly and feel more realistic with a faster digital optical tracking type of system.

INTERFACE. Most of the initial force-feedback devices attach to either the serial port or the game port, and some devices require both (Table 1). Next year's devices will likely migrate to the Universal Serial Bus (USB) available on the next-generation PC motherboards and supported in Windows 98. (USB is similar to SCSI in that devices can be chained off of one port. USB is also faster than the serial port.) The change to USB should reduce feedback latency and the problem with port availability.

Designing Force Effects

The force-feedback element of a game requires aspects of physics and collision detection that should already be implemented and used by the audio engine. When you get down to the code level, force response is very similar to 3D audio. In my experience, piggybacking the audio code helps determine what effects to add, where to add them, and saves both CPU and developers' time by reusing computations. Exceptions to this are effects such as "wheel stiffness," that do not have an associated sound effect, but remain active throughout the simulation.

While Microsoft's DirectX 5 documentation doesn't go into the artistic part of force feedback, the three force-feedback engine manufacturers have a lot to say in their development documentation. I won't cover the design elements in the same manner, so I recommend examining the commercial web sites listed at the end of this article and the books in the "Books On Force-Feedback Technology" sidebar.

The initialization of a force-feedback device is faster with DirectX 5 (if the device is powered), because the device must exist in the game controller's property sheet. Proprietary APIs that don't use this information will search for a force-feedback device on each port and wait for either a confirmation or a timeout before moving on to the next. This can take up to five seconds, which makes automatic detection of a force-feedback device at game start-up less appealing.

Once the force-feedback device is initialized, the user should be able to customize the force settings for the device. At a minimum, users should be able to set the gain, since force-feedback devices tend to vary in the amount of foot-pounds they can exert. (There is a proposal to have this gain setting incorporated into the device's property sheet in Windows, but you'll need to implement this yourself in your game, at least until DirectX 6.) Various types of devices exert different amounts of force (for example, wheels typically exert more force than sticks), but even the forces applied by different brands of joysticks can vary. Gain adjustment may seem like a simple task at first, and it can be depending on how far you take this setting. Gain can easily be applied to Springs and Jolts for example, but Vibrations and custom forces are commonly very temperamental, and adjusting their gain can ruin the desired effect. So, if some effects can be ruined by differences in gain, and force-feedback devices have different force ratings, there's a problem.

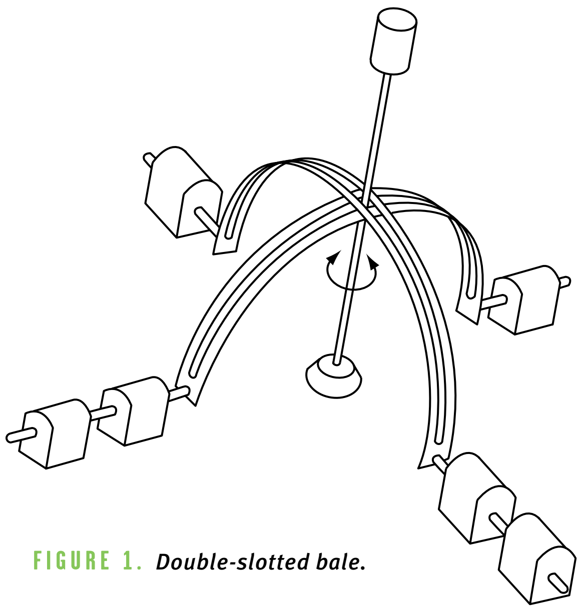


FIGURE 1. *Double-slotted bale.*

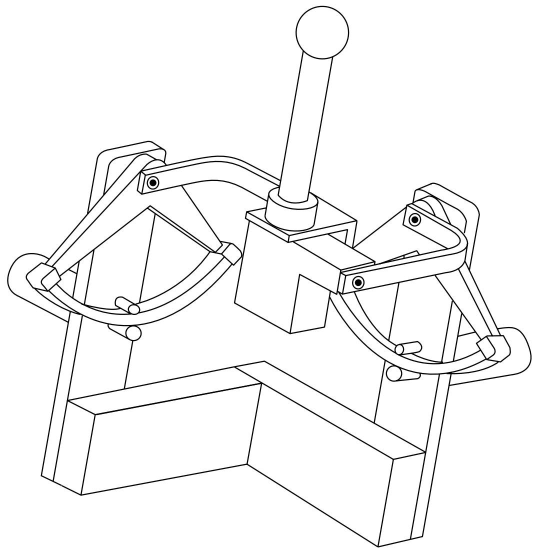


FIGURE 2. *Five-bar linkage.*

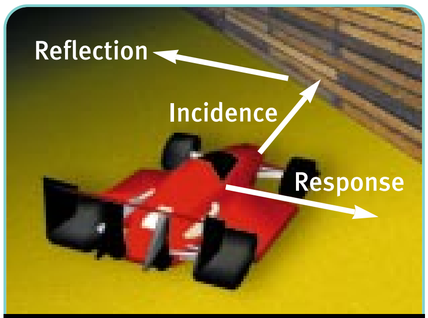


FIGURE 3. *The forces involved in a car crash.*

FIGURE 4. *Microsoft's Visual Force Factory*

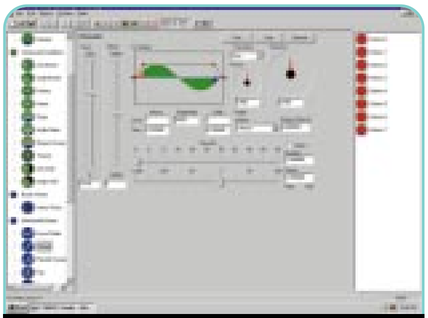


FIGURE 5. *Immersion's I-FORCE Studio*

Basically, each force-feedback device requires some special tuning consideration to get the desired effects just right.

Carefully consider how you allow players to map (or customize) their input controls. When input mappings are altered by the game player, the new mapping should also change the implementation of force-feedback effects. If you don't give adequate thought to how your input device will behave once the player modifies the standard configuration, the player could be in for an unpleasant experience. For instance, imagine that a player is using a joystick in a car racing game. The default mappings have the *y* axis act as a combination gas/brake pedal, increasing the vehicle's speed when the stick is moved forward and braking the car when the stick is moved back toward the player. If the player remaps the *y* axis so that pushing forward on the stick up-shifts the car's gears and pulling back down-shifts the car, then a *y*-axis jolt can cause a problem. A collision to the car from behind would cause a forward jolt effect on the stick, and inadvertently cause the car to shift up a gear.

Allowing the player to turn on or off certain force effects is a nice feature (for example, engine vibration transmitted to the joystick may be annoying in a long race). You could even go so far as to let players customize certain effects. While this may be going overboard, remember that game extensibility is valued by most players. If effects are created with a sensation editor and saved to a file loaded by the game, players may be able to customize these effects with the same editor, and share their customized effects with other players on the Internet.

Switching between various frames of reference in your game is risky. A frame of reference (FOR) is the "who" or "what" to which the force response attaches. You generally only need to consider two FORs, the player's body and their machine (one inside the other, but not rigidly attached). Switching between FORs may confuse the player, making forces appear buggy. For example, forces on a car don't have the same vectors as those on the driver, and switching between the two hurts the realism of the game. Choosing the object that directly interacts with the simulated environment usually provides the best experience. Switching FORs or choosing a bad FOR also complicates development by making it difficult for the engineer to identify and correct undesired feedback loops.

Be careful of feedback loops in your force effects. Feedback loops occur when a force is attractive. A simple example of this is when a car hits a barrier and the frame of reference is attached to the driver of the car, but the collision detection/correction is attached to the car itself (Figure 3).

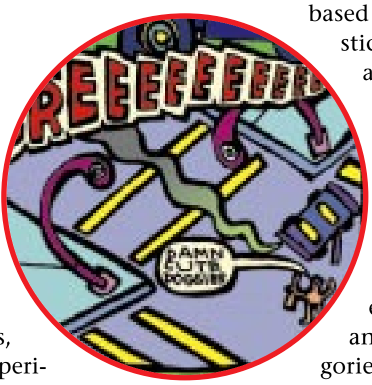
The force response is in the general direction of the barrier, which makes the player hit the barrier again — hence the feedback loop. A feedback loop was not appropriate for the example barrier collision (caused by the poor FOR choice), but in some cases it works out great. Gravity wells, rubber-banding, and tractor beams could all benefit from a feedback loop, which, by the way, can be pulled out of with a little effort on the gamers part.

Types of Feedback Effects

The terms "texture" and "effect" are often used interchangeably to define the force-response data sent to the device. In haptics, "textures" are literally textures (such as sandpaper, wood grain, and so on), so I will refrain from confusing the distinction. Effects can be two-dimensional on joysticks and are one-dimensional on wheels. Many effects are additive, meaning they can be played with other effects simultaneously. Effect addition is nice to a point, but too much can take away from the experience (and wear out the user). The following sections discuss ways of thinking about effects.

STATIC AND DYNAMIC EFFECTS. After creating an effect, it can remain unchanged or undergo continuous modification. Static effects (also known as "canned" effects) don't require runtime modification, which makes them simple and well-suited for trigger effects. A gun shot is a good example of a static effect because it always produces the same kick in the same direction. Static effects can be downloaded and remain on the device (as long as there is enough device RAM), which gives them very low response latency. Static effects are also easier to design because testing doesn't need to be done on numerous variations of the effect, as is the case with dynamic effects. They can also be easily created with the sensation editors (which I will discuss later). Some devices have ROM effects that don't use any RAM (beyond the space allocated for parameter modification); such effects can be used as either static or dynamic effects at a very reduced space cost.

Most effects can be modified during playback, producing a dynamic effect. A joystick vibration that varies with engine RPM is an example of a dynamic effect. Dynamic effects are where force-feedback technology really shines, as they let the player experience the best motion within the game world. Unfortunately, good dynamic effects can also be time consuming to create. One pitfall is that some devices don't work well at the extent of their range; sometimes simply incrementing a variable can drastically change a force effect. For example, some force-feedback devices have too much play and/or motor friction to adequately represent the full spectrum of frequencies for which the engine vibration effect may have been coded. To address this problem, you should critique individual devices to find a general minimum and maximum across the range of force feedback hardware.



OPEN-ENDED AND ONE-SHOT EFFECTS. There are two ways to manage effect playback, and both will probably show up in any given game. One-shot effects are those with a finite duration. They are simple to manage because you play the effect and forget about it. A single jolt to a joystick as a result of pulling the joystick trigger to shoot a gun is an example of a one-shot effect. The effect is executed and stops on its own.

On the other hand, open-ended effects require the game to monitor state information in order to stop the effect at the proper time. Consider, for example, a car (a fast, exotic, red one) driving over a wood-plank bridge. There is no finite duration for the car's time on the bridge, so the effect will need to be monitored by the game to regulate the vibration in accordance with speed, and stopped whenever the car stops or leaves the bridge.

INTERACTIVE AND TIME-BASED EFFECTS. Effects can be divided into interactive and time-based events. Time-based effects are played regardless of where the force-feedback device is positioned. Jolts and vibrations are time-based effects. Interactive effects are based on the state of the stick (position, velocity, and/or acceleration). Springs and friction are some examples of interactive effects. DirectInput categorizes interactive effects as "conditions." All other effects are time-based and fall under the categories of "periodic," "ramp," or "constant" (Table 2).

Force-Feedback APIs

The DirectInput API in DirectX 5 is definitely the force-feedback API of choice in the long run. All three force-feedback engine manufacturers (Immersion, Microsoft, and Cybernet) currently or will soon support DirectInput. That is reason enough to use it, but there are other redeeming qualities. The API is designed to be very flexible and is easily extensible. The problem with DirectInput is that it can be a bit convoluted at first glance.

DirectInput wrappers (whether you make your own or use someone else's) are necessary to restrict the generality and redundancy of the API, which is a result of DirectInput's basis on Microsoft's Component Object Model (COM). Effect management should also be addressed by a wrapper because DirectInput doesn't do this for you. DirectInput didn't hide the RAM limitations of the devices by providing software mixing, which creates a download/offload dilemma that you, the developer, have to address.

Other issues that DirectInput wrappers can simplify are the notation and numeric ranges of the `DIEFFECT` structure variables. For example, the `dwGain` element in the `DIEFFECT` structure has a range of 0-10,000. Gain is analogous to a volume setting for the device, which is easier to manage as a percentage (0-100) of effect strength. A wrapper could also simplify DirectInput's handling of periods. DirectInput uses microsecond notation, which a wrapper could convert to the conceptually easier frequency notation. Finally, get this: direction can be expressed in polar, spherical, or Cartesian coordinates. Decent error handling and recovery can get out of control unless this measurement system is simplified by a wrapper.

Although I recommend using DirectInput as your force-feedback API, it's not your only option. Here's a short description of proprietary APIs and DirectInput wrapper APIs:

I-FORCE 1. This is Immersion's proprietary stand-alone API. I used this simple, easily implemented API in THE NEED FOR SPEED – SPECIAL EDITION. I-FORCE 1 only works with the Immersion engines, but it works in both Windows and DOS. Future effort will likely lean on DirectInput device driver support, rather than improving I-FORCE 1.

CYBERNET 2. The latest Cybernet API supports DirectInput and I-FORCE compatible drivers. It's a stand-alone API that falls between I-FORCE 1 and DirectInput in terms of complexity and features.

WRAPPERS. Using a wrapper will make initial implementation easier because it will hide many of the gotchas. But before you simply jump into using one of these premade wrappers, realize that they are focused around their creators' devices and will exhibit some problems running on their competitors' devices. Immersion and Microsoft both provide source code of their wrappers, and it's an excellent way to determine the differences between the two companies' interpretations of the DirectInput

TABLE 2. *Types of force-feedback effects.*

Group	Category	Type
Interactive	Condition	Spring
		Damper
		Inertia
		Friction
Time-Based	Periodic	Square
		Sine
		Sawtooth Up
		Sawtooth Down
	Constant	Constant
		Raw
	Ramp	Ramp

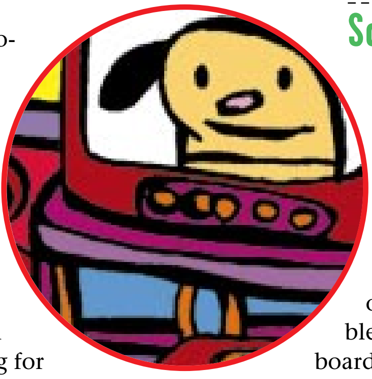
force-feedback specification. It's also a good starting point if you want to create your own universal wrapper. If I decide to use a prefabricated wrapper, I expect it will be for effect file loading only.

I-FORCE 2. This is a misleading name because I-FORCE 2 is not a stand-alone API upgrade to I-FORCE 1. I-FORCE 2 only loads effect files created by Immersion's sensation editor (I-FORCE Studio), and wraps the DirectInput effect create and release interfaces. The I-FORCE 2 documentation has sample code for a more complete wrapper, but this code isn't present in I-FORCE 2.

SIDEWINDER FORCE. This is Microsoft's fully functional DirectInput wrapper. SideWinder Force has functions to work with Microsoft's sensation editor (Visual Force Factory) and wraps effect modification and playback. SideWinder Force also wraps device setup tasks. The source code to the SideWinder Force library is provided with Microsoft's SDK.

Tools

Sensation editors provide a means to visually edit forces. The visual environment helps clarify the purpose of all the variables in a given effect and modify those variables on a picture or graph. After creating an effect, it can be played back, allowing for a fast edit/play development cycle. Once the effect(s) are finalized, they can be saved to a file for the game to load at run time. A force-feedback emulator, on the other hand, creates a virtual device that visually represents force-feedback effects on a monitor.



SENSATION EDITORS. Currently, two sensation editors are available: Microsoft's Visual Force Factory (Figure 4) and Immersion's I-FORCE Studio (Figure 5). Both of these editors output DirectInput-compatible files that can be statically linked to the game or loaded at run time. The benefit of loading the effects at run time is that consumers can modify the effect in the sensation editor themselves. Both editors are capable of single and compound (concatenated) effect creation. They will both support playback within the editor on all DirectInput-compatible force-feedback devices, but the first revisions may not work flawlessly with competing devices due to the infancy of DirectInput force-feedback device drivers. I recommend playing with the editors even if you don't plan on using them extensively. This will help you understand the parameters for various effects.

FORCE-FEEDBACK EMULATOR. Immersion has a serial-based emulator that runs in a DOS window on a second machine. This expedites trouble shooting if you don't have a force-feedback device handy. In some instances, the emulator is better than a device because you can watch the state of a virtual device on your monitor; however, it only works with I-FORCE 1 drivers. This emulator was created when prototype devices were scarce. However, since force-feedback devices are now readily available, both Microsoft and Immersion believe that emulators are non-essential for DirectInput.

Sample DirectInput Code

The sample code on the *Game Developer* web site is straight DirectInput code. I refrained from using wrapper APIs in order to remain flexible. The mouse and keyboard are referenced in the sample code because they were closely tied to the device setup in my test suite. I removed most of this code to minimize the distraction from the joystick and force-feedback code relevant to this discussion.

DEVICE SETUP. The following is an ordered list of essentials to get up and running:

1. Retrieve a pointer to the DirectInput interface (`LPDIRECTINPUT`).
2. Using the `LPDIRECTINPUT`, enumerate the joysticks.
3. Your enumeration callback function can either save the `LPDIDEVICEINSTANCE` passed to it for each joystick for later device creation, or create the device within the callback function (as the sample code does).
4. Using the `LPDIDEVICEINSTANCE` for a given device, get a pointer to the `LPDIRECTINPUTDEVICE` interface for that device.
5. Using the `LPDIRECTINPUTDEVICE`, set the data format (the sample code uses the default format explained in the DirectX 5 documentation).
6. Again using the `LPDIRECTINPUTDEVICE` (`DID`), call `QueryInterface()` to upgrade to an `LPDIRECTINPUTDEVICE2` (`DID2`). The `DID2` inherits all the functionality of a `DID`, so you can even use `DID2` for game controls without force-feedback support. The standard `DID` doesn't support force feedback.
7. Using the `DID2`, set the co-op level. It must be `DISCL_EXCLUSIVE` and either `DISCL_FOREGROUND` or `DISCL_BACKGROUND`.
8. Release the `DID` because you now have a `DID2` that takes its place. This is the last thing I do in my callback function because I have the `DID2` that is used to acquire the device and query its capabilities.
9. Repeat steps 4-8 for each device you want to use.
10. Using the `DID2`, acquire access to the device. This call is made initially and again whenever you lose and regain window focus.
11. Using the `DID2`, get the device's capabilities. This will tell you if the device supports force feedback, as well as how many axes, buttons, and so on, are on the device.

EFFECT SETUP. Ideally, you want the effects that you create and the effects played back in your game to have a one-to-one correspondence. Even though DirectInput makes no attempt to hide the RAM limitations of the force-feedback device, and even though some devices can only have one type of resident effect, it's still possible to get accurate effect playback. You just need to prioritize the playback of effects of the same type.

The basic steps of effect creation are:

1. Determine if the effect is supported by the device.
2. Create the effect using the `DID2` of the device for which you want the effect created (the same effect must be created on each device separately) and passing in a filled in `DIEFFECT` structure.
3. Unload the effect to make room for the next effect.

Effects are automatically downloaded to the device when created. If the device is full, however, the

CreateEffect() interface will produce an error. To prevent this error from occurring, you can set a flag to prevent the automatic download of effects during the effect creation stage, and then turn it back on so that the effects automatically download when you play them.

PLAYBACK. To play a trigger effect, you must download the effect (make sure the trigger button variable is set appropriately in the DIEFFECT structure). Unloading the effect will stop the effect. When the effect is unloaded, pressing the trigger no longer plays the effect. Unfortunately, Immersion and Microsoft have interpreted the DirectX 5 specification for this process slightly differently, and as of this writing they haven't arrived at a common solution.

In Immersion's implementation, starting a trigger effect won't play it immediately. Rather, it will activate the effect to be played when you press the assigned trigger. The benefit of this approach is that multiple effects can be mapped to a trigger and remain resident on the device. This means switching trigger effects is a low-latency process.

In Microsoft's implementation, the trigger effect is ready to use once it is downloaded. If you start the effect, it will play back immediately, even if it's mapped to a trigger and the trigger hasn't been pressed. The benefit to this implementation is that the effect can be used for both trigger and non-trigger effects, but in order to unlink the effect from the trigger, the effect must be modified (requiring download) to detach it from the trigger.

PROCESS LISTS. A process list allows you to concatenate effects so that one plays after the previous is completed. Process lists are being proposed for inclusion in DirectX 6. This feature is currently available through use of the DICUSTOMFORCE effect, which can be created manually (with a fair amount of effort) or with the sensation editors. There is no standard way to concatenate effects, however, so these emulated process lists don't port well between the different force-feedback engines.

MODIFYING EFFECTS. Many effects can be modified during playback, depending on which device you are targeting.

While tuning dynamic effects, be aware of their impact on latency and processor load — you'll run into trouble if you're not making a conscious effort to avoid both. There are several ways to modify effects. You could release the old effect, create a new one, and play it — probably the worst method. The best way is to modify an existing effect with the SetParameters() interface. With this function, you can specify changes in single parameters, and only those parameters will be downloaded,thereby reducing latency.

Another method is to modify an effect and then play or update the effect in the same call to SetParameters(), which reduces the amount of communication required with the device, since the play command piggybacks on the modify command.

MIXING EFFECTS. An effect must be on the device in order to play it. Yet most force-feedback devices limit the number of effects that reside in the device's RAM at any given time, and software mixing isn't supported in DirectInput. The result is that you may be limited to playing only two DICONSTANTFORCE effects at once. Until effect management is improved, you'll probably experience some difficulties with mixing configurations on low-RAM devices.

GAIN ADJUSTMENT. The gain can be adjusted in one of three ways. Most effects have a magnitude parameter for each axis — some go as far as one parameter for each direction of each axis. The second method is to set the gain on the entire effect, which will be applied to all sub-elements, including envelops. The third method works on a device level. All effects are attenuated by the setting passed to SetProperty() (not to be confused with SetParameters(), which works at an effect level).

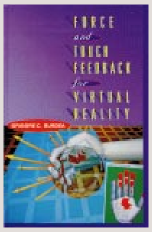
Shake It Up

Force-feedback devices are now readily available to consumers looking for good force response in their games. DirectInput force feedback is a little detailed, but programming with it will increase the chances that all devices will work with your game. Remember that DirectInput force feedback is a "version 1" API, as are the accompanying device drivers. Effect management/emulation is bound to improve in the next revision. Inserting force feedback into the game near the accompanying sound effects will make implementation easier. Competition between the contenders in the force-feedback market is bound to bring out improved force-feedback technologies in the coming years. If all this stops being fun, take a vacation. ■

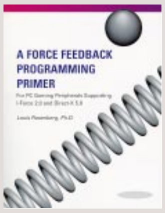
Chuck Walters is a software engineer at Electronic Arts in Seattle, Wash., and received his BSCS degree from the University of Washington, Seattle. He can be reached at cwalters@ea.com.

Books on Force-Feedback Technology

There are two books on force-feedback technology and programming that you should investigate if you want further information on the subject. The first book, Grigore Burdea's *Force and Touch Feedback for Virtual Reality* (John Wiley & Sons, 1996) is a good book about the design and theory of force-feedback hardware.



The second book, Louis Rosenberg's *A Force Feedback Programming Primer*, (Immersion Corp., 1997) is targeted more specifically at the game developer. Rosenberg is the president of Immersion Corp., and his book focuses on the I-FORCE 2 and DirectX 5 methods of implementing force effects. The 177-page book is free and is available an Adobe Acrobat file from the Immersion web site (http://www.force-feedback.com/ pages/ book_request.html).



Back

Real-Time Vo el Te ai Ge e a io

For the past few years, game programmers have been steadily adding terms to their 3D vocabulary — terms such as binary space partition, rasterization, perspective-correct textures, MIP-mapping, and so on. One of these terms has become a catch-all describing a general means to an end rather than the specific structure that it was originally conceived to describe. That word is "voxel," and the context is voxel engines. Voxel engines have come to mean anything that renders terrain-like imagery; whether or not the engine is based on real voxels is immaterial as long as it looks good.

Figure 1 is a screen shot of the voxel engine created in this article compared to a screen shot from the quintessential voxel-based game COMANCHE 3 from NovaLogic. I don't know exactly what technique they're using, but as long as ours looks the same, why ask why? Anyway, in this article, we're going to hold fast with the tradition of using the term voxel loosely to describe a technique used to render data that has volumetric information in it rather than polygonal data. However, before we take off into mathematical hell, let's take a look at real voxel graphics and their data representations.

Real Voxels

Voxel graphics originated in medical imaging. Voxel means "volumetric pixel" and is essentially a cube in 3D space. Each cube has a volume dependent on the length of one of its sides. However, most representations use voxels that are 1 × 1 × 1, so each voxel has a volume of 1 × 1 × 1 = 1.0 voxels. At this point, you may be wondering, "How do I represent voxel data in a computer?" There is no correct answer, but most of the time, voxels are represented by either a 3D array or a more abstract data structure such as a tree. Figure 2 illustrates a voxel geometrically, along with the two most common representations. Voxel data is usually obtained from medical imaging systems such as MRI and CT scanners.

Voxel rendering can be accomplished in a number of ways, such as ray tracing or even

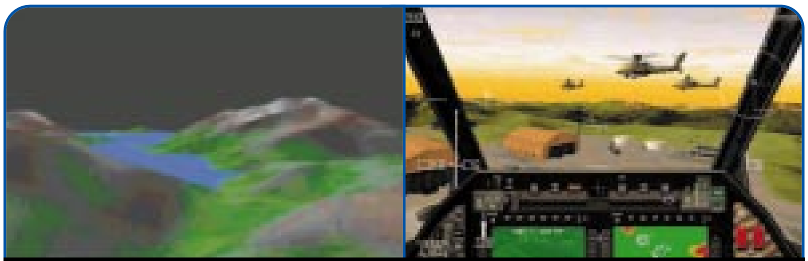


FIGURE 1. *The author's voxel engine is on the left; COMANCHE 3 is on the right.*

with polygon graphics. In the case of ray tracing, imagine that each voxel is a tiny little cube, and a ray tracer traces out the voxel data by computing ray intersections from the viewpoint through the view plane into the voxel data. Of course, this approach would be extremely slow because there are zillions of voxels in even the simplest voxel data set. For example, say that you're scanning your head and you have a scan resolution of 100 dots per inch; that means that there are 100 × 100 × 100 voxels in each cubic inch or 1,000,000 voxels! Now image that your head is about 6 × 6 × 10 (I don't have a ruler) inches for a total of 360 cubic inches — 360 * 1,000,000 = 360,000,000 voxels! At eight bits per voxel, you've got 360 megabytes. Call it half a gigabyte for people with large foreheads, and you've got a real problem. There must be a better way to store and display voxel data. And, of course, there is.

Voxel data can be compressed, take advantage of spatial partitioning techniques, and so on. However, the real problem isn't in the storage, but in the rendering. Processing hundreds of megabytes of data per frame would be a bit slow, to say the least. Therefore, algorithms have been devised that take advantage of the cubic and regular structure of a voxel data set to render the data quickly and in almost photorealistic quality. One such algorithm is the "Marching Cubes" technique, which is based on rendering slices of voxel data. Fortunately, we're not going to learn the marching cubes algorithm, because we don't need to render full voxel data. Rather, we're concerned with data that is partially volumetric, such as height data.

SOME OF THE MOST REALISTIC TERRAINS IN TODAY'S GAMES USE VOXEL TECHNOLOGY. THIS TUTORIAL EXPLAINS VOXELS IN THE CONTEXT OF BUILDING A RUDIMENTARY FLIGHT SIMULATOR *by André LaMothe*

Unreal Voxels

Since we're interested in simply drawing 3D terrain (in real time of course), we don't need full 3D voxel data. A mountain range is basically a 2D grid with a height (and color) at each position, or in another words, a height map (Figure 3). The trick is to figure out a way to render height map data in such a way that it looks 3D and has some resemblance to the data set being rendered. Because we don't want to write a polygon engine that renders little cubes, or more precisely, vertical parallelepipeds, we're stuck with using some form of ray tracing to create our display.

This isn't as bad as it seems. Just as ray casting (the technique used for WOLFENSTEIN and RISE OF THE TRIAD) works for worlds made of rectangular solids, we can use ray casting to render worlds that are composed of rectangular solids that

FIGURE 2. *Three representations of a voxel.*

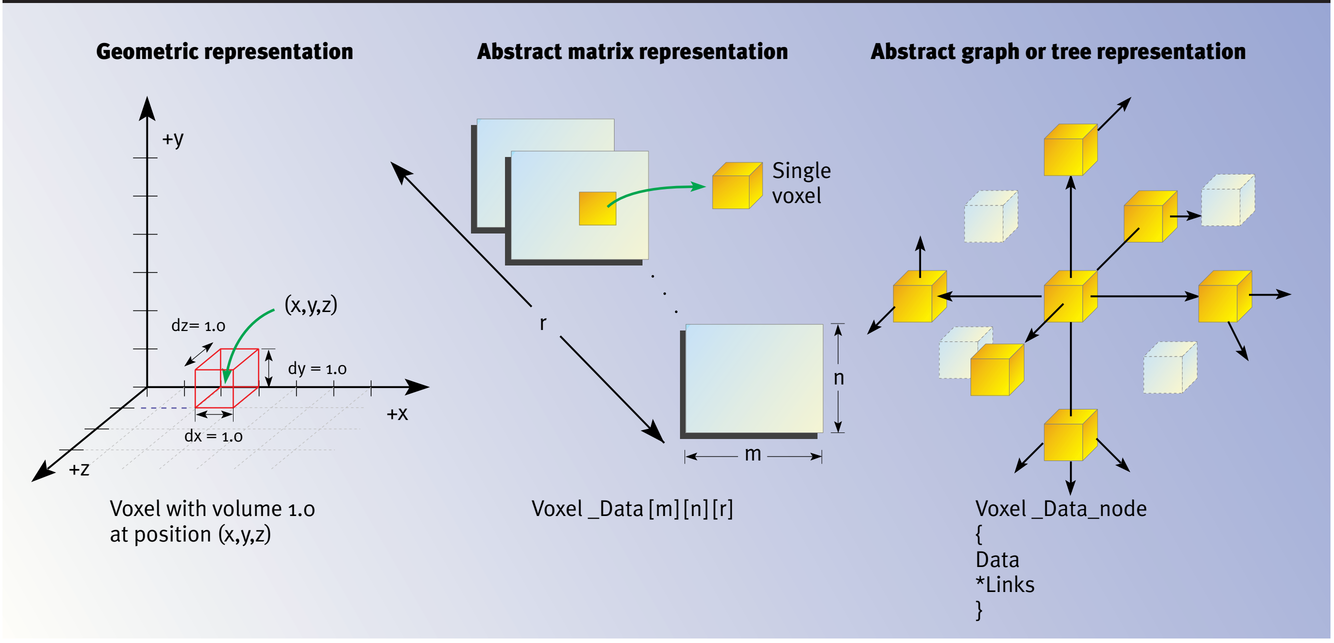
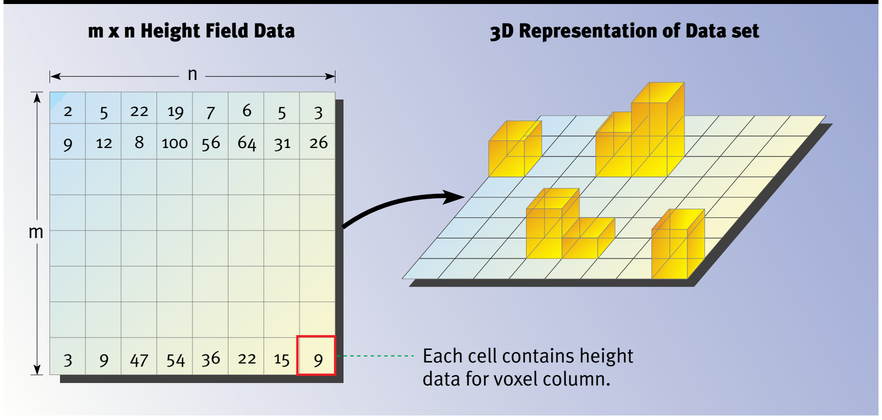


FIGURE 3. *Height map data.*



just happen to be about a pixel or two in thickness. All we need to do is change the scale of our thinking and use the same idea. In addition, because we're going to use height map data, we don't need a ton of 3D voxels to describe our data set. We simply need a 2D array that contains a height in each cell. Therefore, we've already killed at least one full dimension of storage — and that is a good thing.

There is once catch to our representation/compression technique; the minimum altitude of all heights is the same (that is, sea level), and each column must have the same color along its vertical extent. Of course, we could add another 2D array that contains the starting altitude, allowing us to render caves, but we'll keep it simple this time around. The real bummer is that each voxel column in the height map must be the same color. However, if you think about it, this doesn't matter, because when you're looking at a mountain range, you only see the top of each vertical column. The color of the material at various depths is irrelevant unless you have x-ray vision and can see through the mountain material. So the bottom line is that all we need is one 2D map to represent the height data, and maybe another 2D map, similar to a texture map, that represents the color for each voxel column. So if the world is 1,000 × 1,000, then we need a total of 1MB × 2 or 2MB of storage, assuming a byte for each height and color. That's a pretty good savings from our original 500 Mb of full 3D voxel data!

FIGURE 4. *The 3D viewing system.*

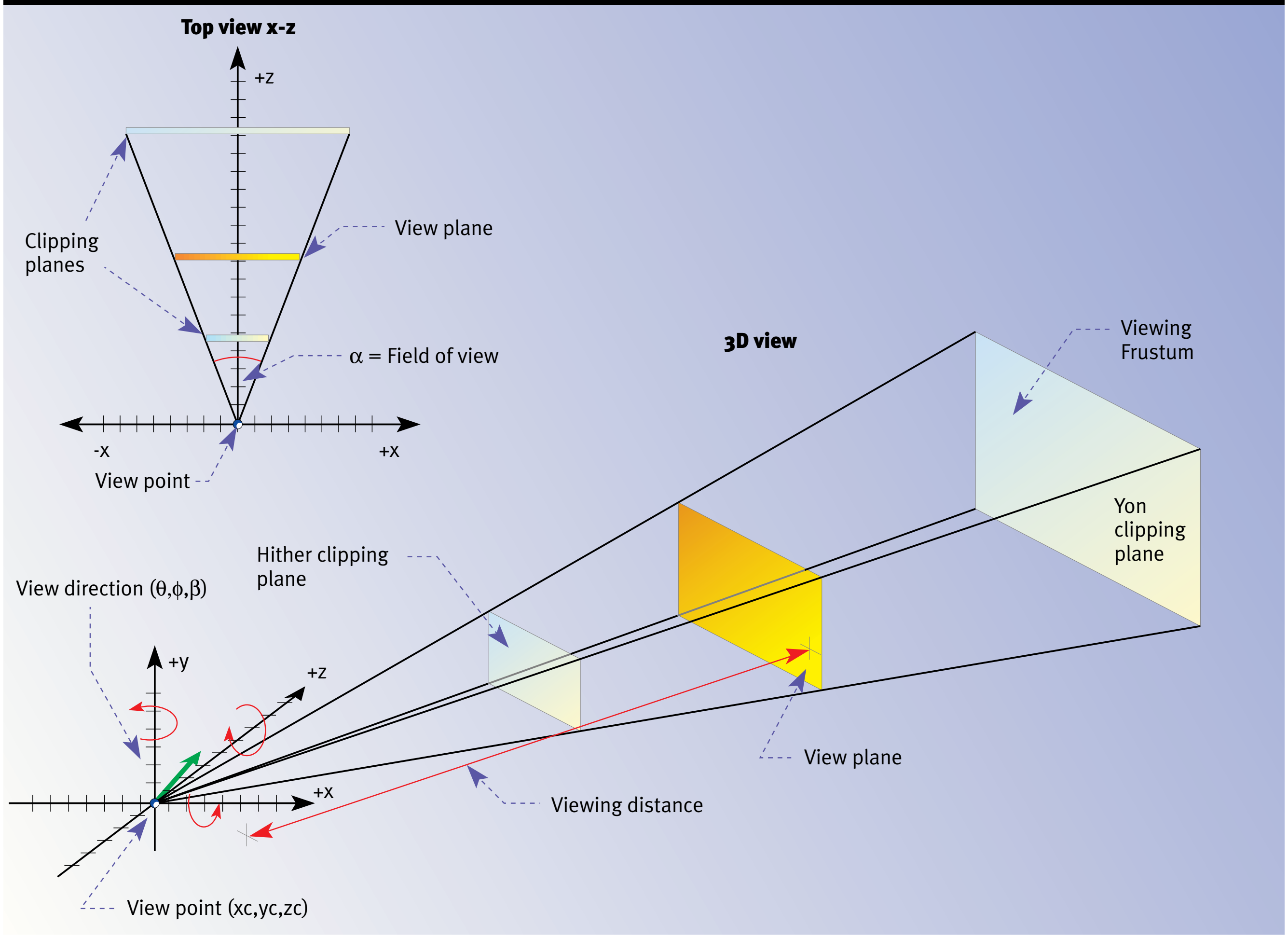
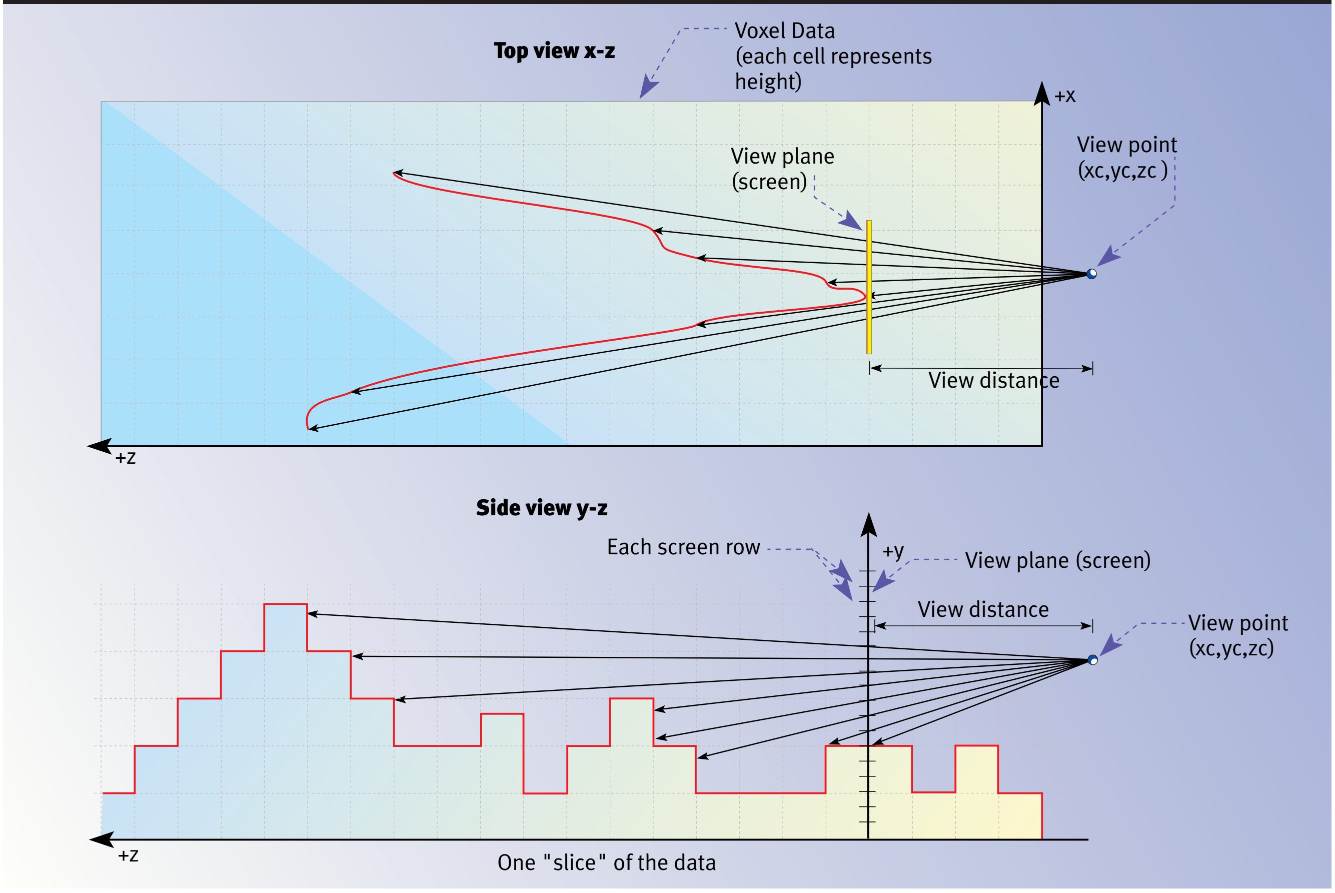


FIGURE 5. *Ray tracing voxel data.*



A Ray Casting Terrain Algorithm

Before we write the terrain generator, we're going to look at how to do it the slow way and then we'll make a number of optimizations to make it about 100 to 1,000 times faster. The key here is to understand the geometry and what is happening in the terrain generator. Don't worry, we're not going to plunge into gradients and vector calculus. We'll use nothing more than basic geometry with a little bit of trigonometry.

Let's begin with some terminology — refer to Figure 4 as we go. The position that we're going to view the scene from is called the "view point," or "camera." It has a position (xc,yc,zc) and orientation (θ, ϕ, β). The position is in "world coordinates," and the orientation coordinates are angles analogous to the pitch, yaw, and roll of the camera. The direction that the camera is pointing is called the "view direction." The object we're looking at is called the "view plane," which is a surface representing the window onto which we're mapping the 2D information to generate the illusion of 3D. The view plane is usually the screen, but doesn't necessarily have to be. Also, the perpendicular distance from the view plane to the camera is called the "view distance." Finally, the camera has a "viewing volume," or "frustum." This is the area that is visible to the camera. For example, most people have about a 140-degree horizontal field of view (FOV) and a 90- to 120-degree vertical FOV. So we should make our computer models within the same order of magnitude if they are to look at all real. For example, DOOM and QUAKE use a 90-degree FOV. A game with a 60-degree FOV would show less, and a 180-degree FOV would be like looking out of a wide-angle lens — that is, lots of distortion.

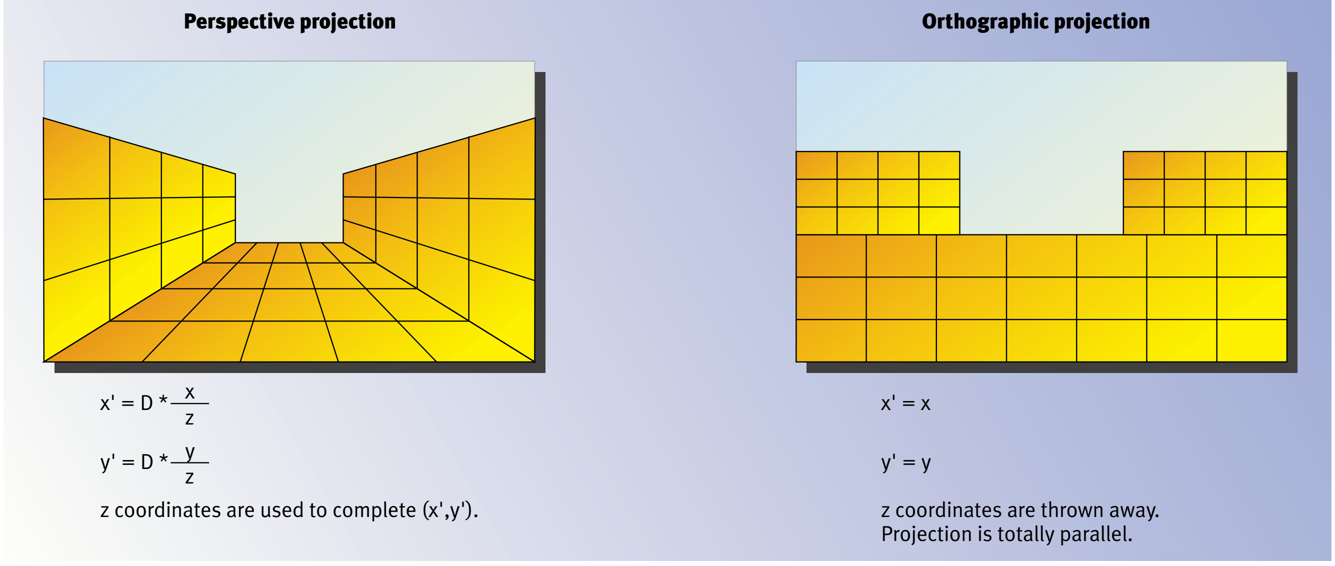
Imagine that the camera is flying above the terrain at some height, and we want to see what the resulting image would be on the computer screen. This can be accomplished by brute force ray tracing. For every pixel on the screen, we'll construct a ray that originates from the camera, pierces the screen (at the pixel), and continues on until it hits the terrain. Figure 5 illustrates this using multiple views. Listing 1 shows the pseudo-code.

That's all there is to it. Because of the algorithm's design and its close analogy to the physics of photons and real

LISTING 1. *Pseudo-code for brute force ray tracing.*

```
For (each pixel on the screen)
   {
   1. Compute ray originating from camera thru pixel.
   2. Project the ray.
   3. If (Projected ray intersects terrain)
        {
        Color screen pixel color of intersected terrain data.
        } // end if

   } // end loop
```

FIGURE 6. *Screen projections.*



$x' = D * \frac{x}{z}$

$y' = D * \frac{y}{z}$

z coordinates are used to complete (x',y').

$x' = x$

$y' = y$

z coordinates are thrown away.
Projection is totally parallel.

light, the scene will be rendered almost perfectly (of course, there won't be shadows or reflections). The only problem is that Steps 1, 2, and 3 take a lot of computer cycles. For example, if you want to render a screen image that is 320 × 200, you'll have 64,000 pixels. Hence, we must compute 64,000 rays or parametric lines, project these until they intersect with the terrain data, compute the points of intersection, and then plot the pixels on the screen in the correct color. That's a lot of computing. We need a way of doing the same thing with less math and fewer computations. This can be accomplished by taking advantage of the regularity of the data and the behavior of perspective transforms.

Optimizing the Algorithm

We're now going to derive an algorithm and associated optimizations so that voxel terrain data can be rendered at faster rates. However, the algorithm is so simple, you really need pay attention to all the subtle details. Let's begin by taking a look at the derivation of the perspective transform and what it does for us.

When drawing 3D computer graphics on a 2D computer screen, we only have two dimensions. Therefore, to convey the data that is lost in the translation of 3D to 2D, various projection techniques have been created. One such technique, perspective projection, is based on the premise that as an object moves away from the view plane and camera, it should get smaller. Similarly, as the same object gets closer to the view plane and camera, it should get larger. Therefore, we can deduce the projection from the z coordinate of each vertex making up a 3D object.

Figure 6 represents a perspective transformation as well as a nonperspective, or orthographic, projection. As you can see, the perspective projection looks a lot better — it looks 3D. The key to generating a perspective projection is using the z component of each vertex to scale the position of each (x,y) to arrive at a new (x,y). It might be easier to understand this relationship by looking at how real light works. Take a look at Figure 7a. Here we see a line segment at some position on the left side of the view plane. To generate the perspective projection of the line on the view plane, all we need to do is draw rays from the view point through the view plane to each end of the line segment. The point where each of these lines intersects the view plane is the perspective-correct line that we would see if we were viewing the scene at the given viewing distance through a window.

Figure 7b shows the derivation of the perspective transform. The figure shown is a side view of the z-y plane, so there isn't any x information; a similar derivation can be done for the z-x plane. Anyway, the view point lies on the z axis at a distance V_d from the view plane, and the point P = (Px,Py,Pz) is at some distance Pz and at some height Py from the view point. Finally, P' = (Px',Py',Pz') is the point of intersection on the view plane. Using similar triangles, we extract the relationship

$$\frac{Py}{Py'} = \frac{Pz}{V_d}$$

or,

$$Py' = V_d * \frac{Py}{Pz}.$$

Thus, it looks like the perspective transform of a point (x,y,z) at a viewing distance of V_d is

$$\text{x\_perspective} = (V_d * Px/Pz)$$
$$\text{y\_perspective} = (V_d * Py/Pz).$$

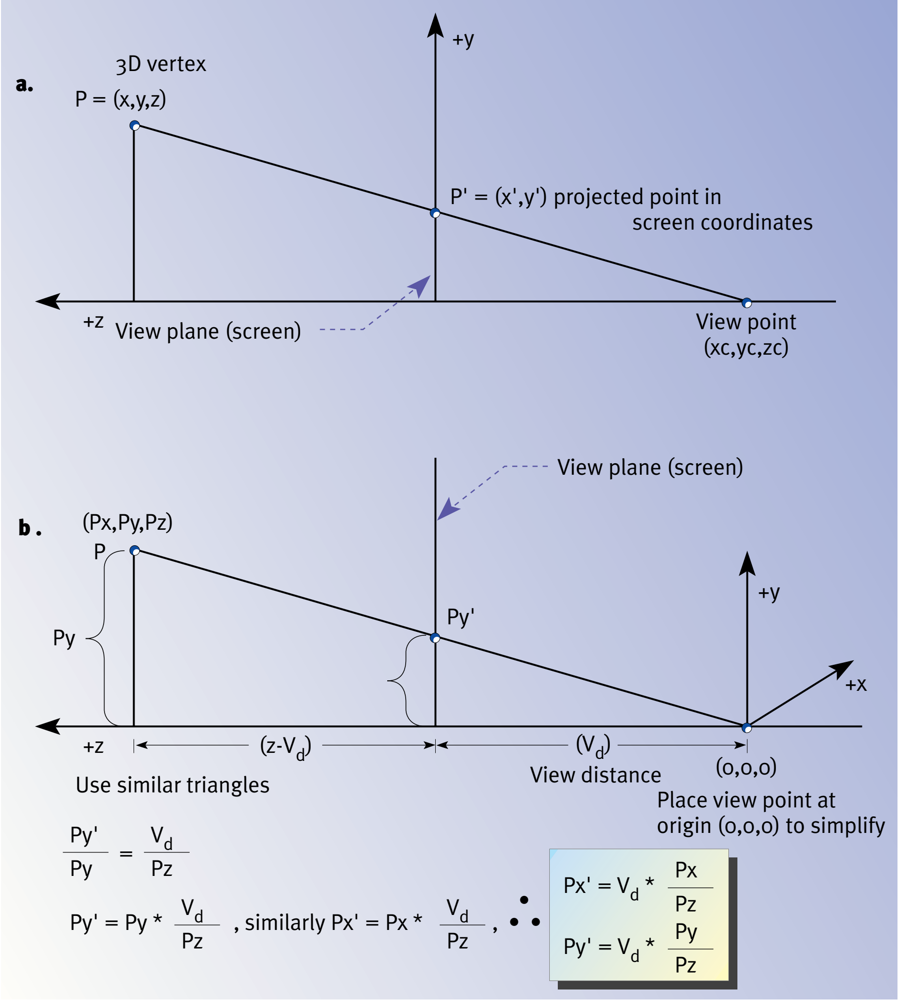
This tells us a couple things: First, the height or size of a line decreases as it gets farther away from the view plane. We therefore know that the scale of an object is also proportional to the z distance that it is from the view plane. We're going to base our entire algorithm on this important fact.

Now it's time to get down to the details. We have all the tools that we need to really analyze the situation and come up with a better algorithm than the brute force method of computing a ray for each pixel on the screen. What we're going to do is trace out, or contour map, each vertical scanline of the screen by first casting out a single ray for each column on the screen. Based on the first voxel column that the ray hits, we'll then follow the contour of the terrain from that point on without any further rays being cast out in that particular column (Figure 8).

But before you do that, let me offer you a warning. The following figures are a bit cavalier in their labeling of the axes. This is due to the fact that we're trying to mix world coordinates, polar coordinates, and screen coordinates. The bottom line is that when we finally get to writing the engine, the x-y axis will be the ground plane, and the z axis will vertical. However, for a number of the figures, I used z to go into the screen and y for vertical just to make things easier to understand.

It's possible to find the first intersection of a ray cast from the view point through the bottom of the view plane into the terrain, and then follow the rest of the terrain's contour and render it based on the math and consistency of the perspective transform. How? Well, we need to make a few observations before we can construct the algorithm. Figure 9 shows the view point with a number of rays being cast through each pixel in the same screen column. This figure contains a lot of information, so let's review it slowly.

FIGURE 7. *The derivation of the perspective transform.*



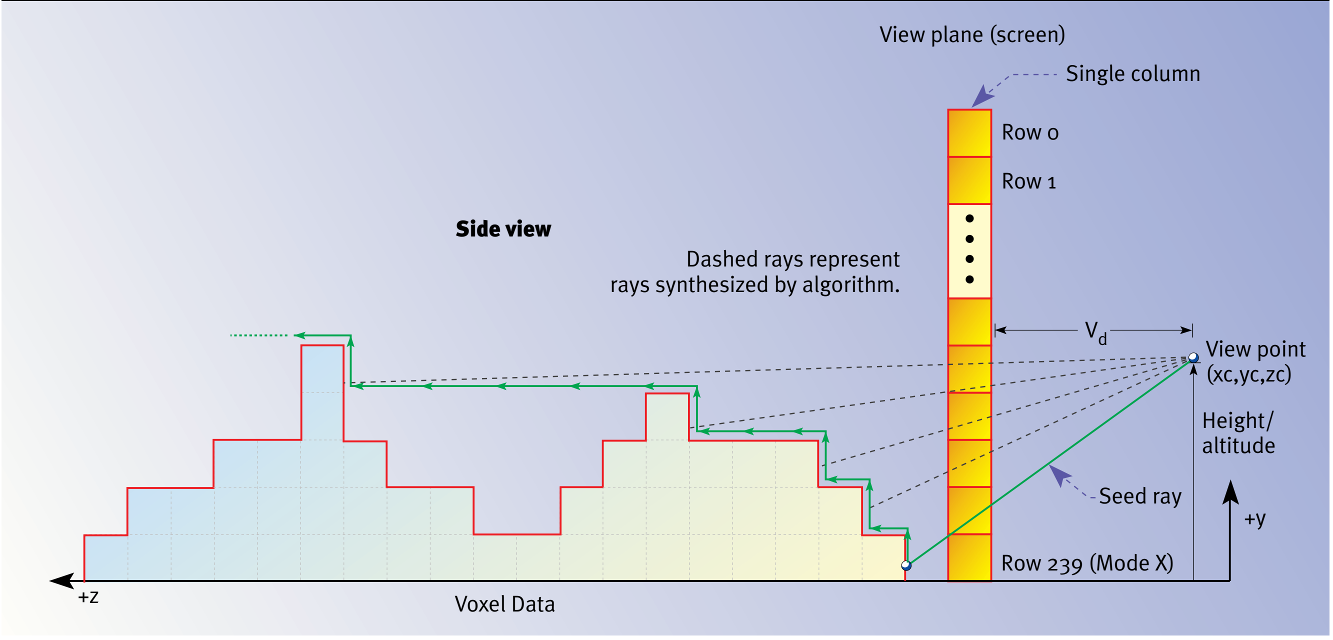
$$\frac{Py'}{Py} = \frac{V_d}{Pz}$$

$$Py' = Py * \frac{V_d}{Pz}\text{, similarly } Px' = Px * \frac{V_d}{Pz}\text{, } \therefore$$

$$Px' = V_d * \frac{Px}{Pz}$$

$$Py' = V_d * \frac{Py}{Pz}$$

FIGURE 8. *Contour ray casting.*



The first thing I want to bring to your attention is the slope of the lines. The first line has a slope m_1. Remember that

$$\text{Slope} = \frac{\text{Rise}}{\text{Run}}.$$

And remember, slopes don't always have to be in terms of x and y — they can be in terms of anything. A slope is the rate of change of one variable in relation to another. In our case, we're interested in the rate of change of z with respect to x or dz/dx. In Figure 9, the first line with slope m_1 has dz = 1 and dx = V_d. Therefore, the slope must be

$$m_1 = \frac{dz}{V_d} = \frac{1}{V_d}.$$

That's interesting. The slope of a line that originates at a distance V_d from the view plane and pierces the first pixel below (or above) has a slope of $1/V_d$ (the reciprocal). If you're with me, then hold onto your hat, because here's the clincher; in addition to m_1 having slope 1/dz, m_2 has a slope of $2/V_d$. Hence,

$$\begin{aligned} m_1 &= 1/V_d \\ m_2 &= 2/V_d \\ m_3 &= 3/V_d \\ &\vdots \end{aligned}.$$

Or, in terms of m_1,

$$\begin{aligned} m_1 &= 1 * m_1 \\ m_2 &= 2 * m_1 \\ m_3 &= 3 * m_1 \\ &\vdots \end{aligned}.$$

What this means is that the change in slope per vertical screen pixel, or per dy, is constant and equal to $1/V_d$, which we'll call ds. Referring again to Figure 9, we see that the lines with slope $m_1, m_2, \ldots m_5$ each intersect lines parallel to the view plane at constant intervals. In other words,

$$\begin{aligned} y_1a &= 1 * ds * (V_d + z_a) \\ y_2a &= 2 * ds * (V_d + z_a) \\ y_3a &= 3 * ds * (V_d + z_a) \\ &\vdots \\ dy_a &= ds * (vd + z_a) = ds * \text{constant}_a, \end{aligned}$$

and

$$\begin{aligned} y_1b &= 1 * ds * (V_d + z_b) \\ y_2b &= 2 * ds * (V_d + z_b) \\ y_3b &= 3 * ds * (V_d + z_b) \\ &\vdots \\ dy_b &= ds * (V_d + z_b) = ds * \text{constant}_b. \end{aligned}$$

So the points of intersection of a ray along any vertical line at a given distance from the view point are also at constant intervals. Moreover, this constant increases at a rate of (ds*z), where z is just the distance between the line and the view point along the z axis.

That's all there is to the algorithm. We can now create an incremental ray caster that casts a single ray for each column of the screen and then generates a vertical column of pixels that represent what the display would have looked like if we had cast a ray for each pixel in the column. Let me repeat, the algorithm we're going to come up with only casts a single ray from the view point through the bottom of the view plane (the screen) for each column. The ray's intersection is then computed, and from that point on, the rest of the contour for that column is generated incrementally using the math that we've derived.

That sounds great, but how? Our plan of attack is as follows: First, we'll need to generate a ray that has a dx,dy, and dz for each column of the screen. We've talked about the dz part, but what about dx and dy? The dx and dy components are computed by finding the endpoints of a unit vector that lies in the ground plane and is perpendicular. For each column of the screen, we want to cast a ray. So for a screen that is 320 pixels wide, such as Mode X, we need to cast 320 rays, each at a slightly different angle. To do this, we first need to decide on our horizontal FOV. Assuming that we want a 60-degree FOV and that our screen is 320-pixels wide, we'll need to break 360 degrees up into smaller subangles so that from any viewing direction (that is, heading), we can cast out 320 rays: 160 to the left and 160 to the right. So we create virtual degrees, based on the FOV and the screen width. The formula is

$$\text{Number of Virtual Degrees} = \text{Screen Width} * \frac{360}{\text{Field of View}}.$$

In our case, this is equal to 320*(360/60) = 1,920 virtual degrees. With that computation out of the way, let's generate the dx,dy part of the ray

FIGURE 9. *The geometry of height mapping.*

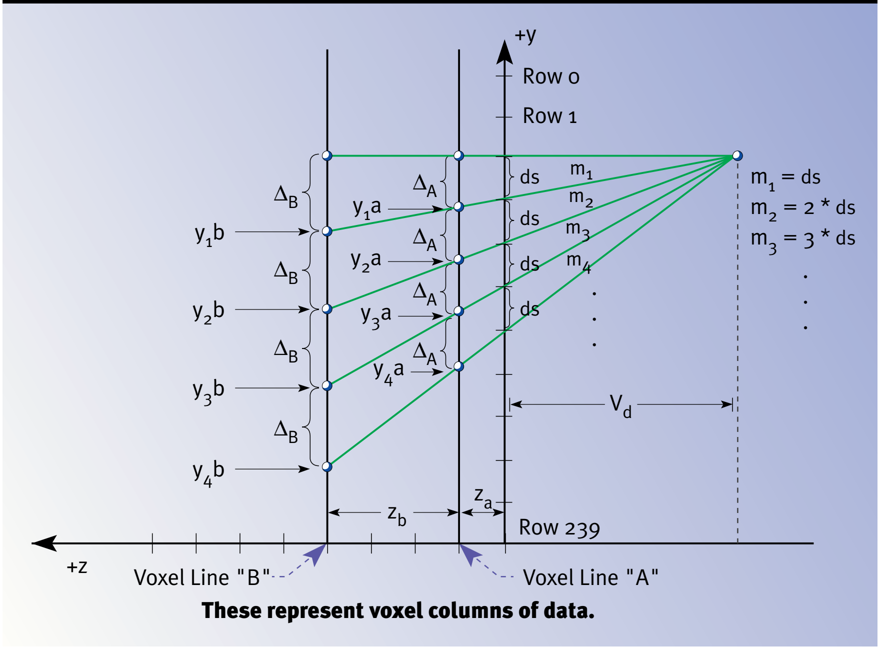
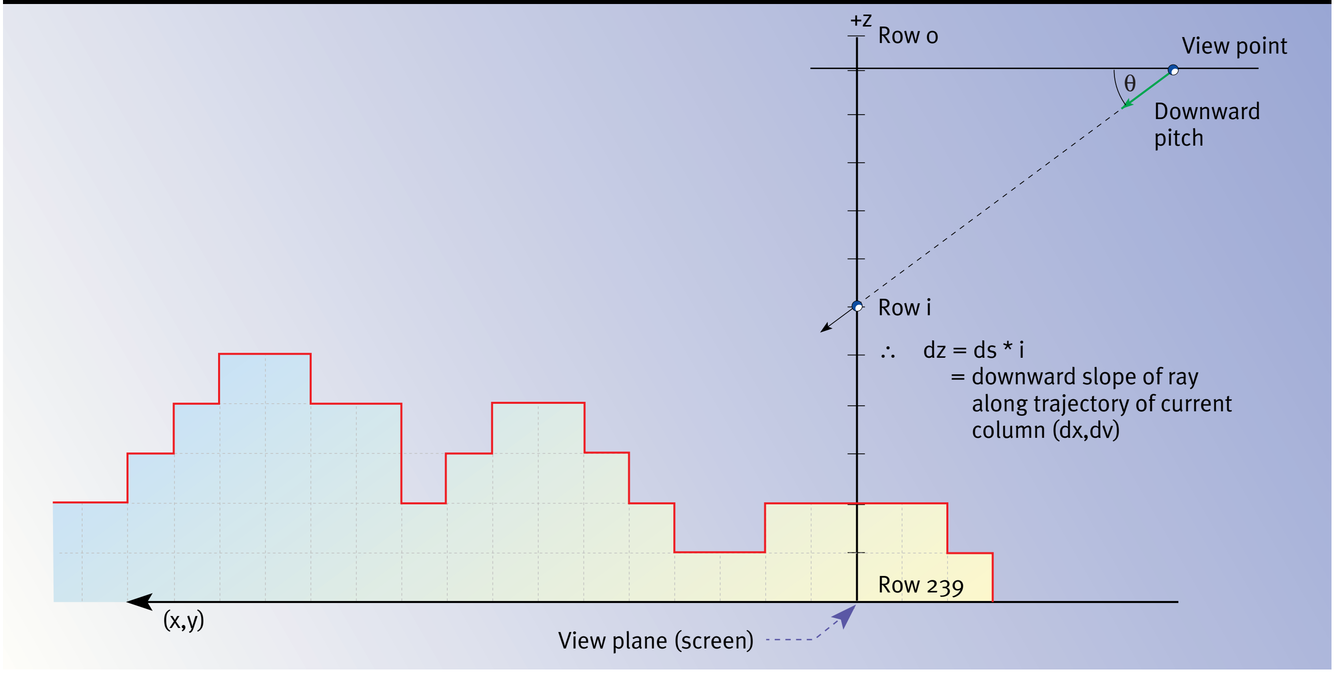


FIGURE 10. *Computing dz.*



trajectory. From basic trigonometry, we know that any point on a circle with radius r is equal to

$$x = r * \cos(\theta)$$
$$y = r * \sin(\theta)$$

where θ is the angle made with the positive x axis.

Since our camera is always flying above the terrain at an altitude of z, we can use this equation to generate our dx,dy deltas. However, instead of using the standard sine and cosine, we must use versions of them that can take angles from 0 to (1920 - 1). That's why we'll need to make look up tables. The pseudo-code for our outer loop is in Listing 2.

Now back to the computation of dz. You'll recall that dz is the slope of the downward ray that we are casting to compute the very first intersection (Figure 10). Referring to the figure, the downward pitch of the camera relates to the starting position of the ray cast. This means that if the camera is pointing down to row 100 of the screen, then the slope of the ray is 100*ds. Therefore,

$$dz = \text{screen row} * ds$$

Of course, since the screen's y coordinate gets more positive on the way down, we'll need to alter this equation a bit; in fact, we'll need to do a lot of little tweaks to make things work.

Now that we know how to compute dx,dy, and dz, we're finally ready to complete the algorithm. Once we compute these values, we enter a loop to render the current column. The loop will project the ray from the camera's view point (xc,yc,zc) at a rate of dx,dy,dz. However, we must track one other variable — the "current projected scale." Recall that we derived that the deltas between ray intersections piercing adjacent pixels in a single column will intersect any vertical column beyond the view plane at constant intervals. These intervals are equal to the current distance from the view point multiplied by the change in slope between adjacent pixels on the screen in a single column, which is always the same and is called ds in our derivation. This means that if you're some distance d from the view plane and you're trying to draw a voxel column, then you need to scale the column by this amount to take into consideration the perspective scaling that exists at the given distance from the view plane. And the amount of scaling or difference between adjacent intersections in a vertical column gets larger as we move farther away from the view plane.

Now that we know that we need to track this "scaling" variable, how do we use it? When the main rendering loop is entered, the position of the ray is updated with the deltas dx,dy,dz, and then the current z position of the ray is compared to the height data located in the height map at the current ray's downward projected (x,y). If the height in the height map is higher than the ray, we enter a second inner loop. In this inner loop, the goal is to draw pixels in the column until we have drawn enough pixels so that the perspec-

LISTING 2. *The outer projector loop.*

```
// start ray off at far left column
curr_angle = curr_heading + 160

for (int column = 0; column < 320 column++)
    {
    dx = COS_LOOK[curr_angle];
    dy = SIN_LOOK[curr_angle];
    dz = ?

    // draw the column

   // adjust the trajectory angle, rotate ray to right
   curr_angle--;

   } // end for
```

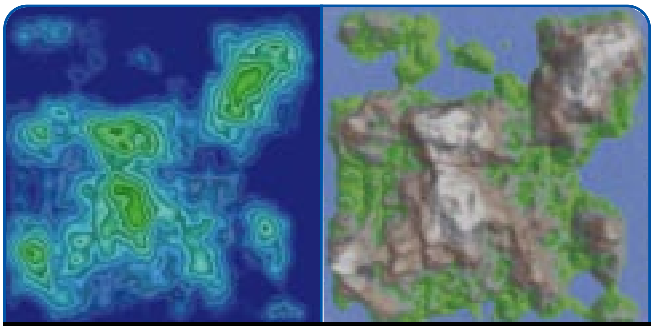


FIGURE 11. *Using VistraPro 4.0 to render terrain. 11a (left) is the terrain height data exported as a 256-color gradient contour map. 11b (right) is the rendered terrain based on that contour map.*

tive projection of the voxel column looks correct on the display. This is accomplished by having an exit strategy based on the current projected scale. At each iteration, we add the current projected scale to the z of the ray, which has the effect of "climbing" up the voxel column. At the same time, we adjust the slope of the ray, because each pixel we draw means that the new ray would have a slope change of ds. Therefore, dz is incremented by ds. Finally, when z is above the voxel column (or when we've hit the top edge of the screen), we bail out of the loop and continue tracing the contour. The tracing process must be done for a number of steps from "hither" to "yon" in Figure 4 to cast far enough to see reasonable detail.

Putting It All Together

Now to write a program that uses our algorithm and generates the 3D voxel display. Here is where all the details come into play. Before we dive into the voxel display generator and the main control module, let's cover a quick check list of what our code needs to do. First, we need to generate height data and a color map, and we need to encode them in some common format such as .BMP or .PCX. The height data can be generated algorithmically, or you can download satellite height data — or if you have an extra $100.00, you can get a copy of VistaPro 4.0 (http://www.callamer.com/vrli/vp.html), which is what I did. VistaPro is a fractal terrain generator that creates height data and 3D views with full texturing of the height data. Thus, you can create any fractal landscape you wish. The program allows you to export the terrain height data as a 256-color gradient contour map (Figure 11a). We can read in this file and use the color index as the height of each voxel column.

Getting the texture information is a little tricky, but basically what you do is place the camera at a (x,y) position in the center of your world with a z altitude of 10,000 units or so. From this altitude, the 3D terrain looks like a 2D texture map. You can export it from VistaPro and then you can use this texture data as the texture color information for your rendering (Figure 11b). So what size should we make the height and color maps? I started off with 1,000 × 1,000-pixel height maps and then realized these files would be too big to download — about 2MB for everything. So I decided to go with 512 × 512 height and texture maps. This works out fine, looks good, and all the data for two worlds is less than 400k after compression.

Now that we know the how and what of the data, let's talk about the program. I've written two versions of the voxel terrain generator program; one with DirectX for Windows 95 (VOXELWIN.*) and the other in DOS 32 protected mode (VOXELD32.*). They're both way too long to list in their entirety here, so we'll only cover the main voxel rendering module and the main control module. Let's begin with the terrain-rendering function `Render_Terrain(...)`.

The function implements our algorithm almost verbatim. The only difference is that everything is in fixed-point math with a 20.12 format — this, of course, is for speed. While the floating-point processor may be faster than the integer processor for multiplication, when you convert floating-point values back to integers you're looking at 40-60 cycles per float. That's horrible. The key to rasterization algorithms is to convert the input into integers or fixed-point and, for the entire algorithm, perform all pixel rasterization with integers — don't mix integers and floats. You can use both, but keep them separate, or the implicit conversions and casts will kill your performance. Anyway, let's briefly cover the function and see what it does.

First, you must send the function the (x,y,z) world coordinates of the view point along with the pitch, yaw, and roll of the camera. Actually, the roll isn't used at all, and the pitch is really more the horizon than an angle. The last parameter is the destination buffer into which the scene frame is rendered. The scene frame is defined in the main program's define section to be 320 × 240. Once the function is entered, the first

LISTING 3.0. *The Terrain Rendering Algorithm.*

```
void Render_Terrain(int vp_x,
                    int vp_y,
                    int vp_z,
                    int vp_ang_x,
                    int vp_ang_y,
                    int vp_ang_z,
                    UCHAR *dest_buffer)
{
// this function renders the terrain at the given position and
// orientation

int xr,             // used to compute the point the ray
    yr,             // intersects the height data
    curr_column,    // current screen column being processed
    curr_step,      // current step ray is at
    raycast_ang,    // current angle of ray being cast
    dx,dy,dz,       // general deltas for ray to move from
                    // pt to pt
    curr_voxel_scale, // current scaling factor to draw each
                    // voxel line
    column_height,  // height of the column intersected and
                    // being rendered
    curr_row,       // number of rows processed in current
                    // column
    x_ray,y_ray,z_ray, // the position of the tip of the ray
    map_addr;       // temp var used to hold the addr of data
                    // bytes

UCHAR color,        // color of pixel being rendered
    *dest_column_ptr; // address screen pixel being rendered
```

thing we do is to convert everything to fixed point and compute the starting video address and ray casting angle.

Now that the starting position of the ray is computed, along with the deltas, the main stepping loop is entered, which will step out on each ray trajectory about 200 times. This procedure is also controlled by a **#define** in the main program. Now, for each iteration of the main stepping loop, the height of the voxel under the ray being cast is tested to see if it is higher than the current ray's z value or height. If so, the voxel column segment **while(...)** loop is entered, and the voxel column is drawn until its projection is as large as its height. Then the **while(...)** is exited, and the ray's position, along with the current voxel scale, is updated with the appropriate deltas. The function can literally be ripped out and used in any program as long as you include the **#define**s and the main global data access pointers **height_map_ptr** and **color_map_ptr**, which point to the data for the height field and texture map respectively. Let's see the function in action with a complete application (a really simple flight simulator).

The Killer App: A 1,000-Line Flight Sim

My original goal was to create a demo that allowed you to fly around the terrain data and change the viewing parameters. However, I got tired of controlling the flight, so I decided to write a little bit of AI to control an autopilot that would fly over the terrain when we left the controls alone for a couple of minutes. I originally wrote the demo in DirectX and Windows 95 (using Visual C++ 4.0), then I decided to port it to protected mode DOS (using Watcom 10.6), because half of the programmers that I know still don't have DirectX on their computers, let alone the SDK. (I never thought I would be porting Windows programs back to DOS.) All of this software is on the *Game Developer* web site.

To run to the DirectX version, you'll need the DirectX 3 or better run-time DLLs, and to run the DOS version, you'll only need the DOS extender, which is included in the files. Both demos will load the height and color data by default, so all you need to do is execute either application. If you want to supply your own height and color data, then the command line parameters are

- **VOXEL(WN1|WN2|D32).EXE height_data_file.bmp color_data_file.bmp**
- Both files must be on the command line.
- Both file names must be in 8.3 format.
- Files must be in 512 × 512 in 256 colors.
- Files must be .BMP format with no compression.

The main control module of the voxel engine is called **Game_Main(...)**. **Game_Main(...)** isn't too complex, but I want to peruse briefly its operation. The function is called once per frame and is responsible for taking the input, altering the flight model (or lack thereof), and rendering the terrain. You have control of the view point, its orientation, and the speed of motion. Also, the autopilot that I mentioned earlier will engage in about one minute if you don't touch any keys; to regain control, turn left or right. The autopilot selects random speeds, turns, and follows the height of the terrain, so it's a cool screen saver. The DirectX version of the code (also on the *Game Developer* web site), has a couple of Windows calls, such as the frame synchronization and the keyboard access. Otherwise, its no different from the DOS version.

LISTING 3.0 (CONT.). *The Terrain Rendering Algorithm.*

```
// convert needed vars to fixed point
vp_x = (vp_x << FIXP_SHIFT);
vp_y = (vp_y << FIXP_SHIFT);
vp_z = (vp_z << FIXP_SHIFT);

// push down destination buffer to bottom of screen
dest_buffer += (SCREEN_WIDTH * (SCREEN_HEIGHT-1));

// compute starting angle
raycast_ang = vp_ang_y + ANGLE_30;

// cast a ray for each column of the screen
for (curr_column=0; curr_column < SCREEN_WIDTH-1; curr_column++)
{
// seed starting point for cast
x_ray = vp_x;
y_ray = vp_y;
z_ray = vp_z;

// compute deltas to project ray at, note the spherical cancelation
// factor
dx = COS_LOOK(raycast_ang) << 1;
dy = SIN_LOOK(raycast_ang) << 1;

// dz is a bit complex, remember dz is the slope of the ray we are
// casting, therefore, we need to take into consideration the
// down angle, or x axis angle, the more we are looking down the
// larger the intial dz must be
dz = dslope * (vp_ang_x - SCREEN_HEIGHT);

// reset current voxel scale
curr_voxel_scale = 0;

// reset row
curr_row = 0;

// get starting address of bottom of current video column
dest_column_ptr = dest_buffer;

// enter into casting loop
for (curr_step = 0; curr_step < MAX_STEPS; curr_step++)
    {
    // compute pixel in height map to process
    // note that the ray is converted back to an int
    // and it is clipped to to stay positive and in range
    xr = (x_ray  >> FIXP_SHIFT);
    yr = (y_ray  >> FIXP_SHIFT);

    xr = (xr & (HFIELD_WIDTH-1));
    yr = (yr & (HFIELD_HEIGHT-1));

    map_addr = (xr + (yr << HFIELD_BIT_SHIFT));

    // get current height in height map, note the conversion to
    // fixed point and the added multiplication factor used to
    // scale the mountains
    column_height = (height_map_ptr[map_addr]
                   <<(FIXP_SHIFT+TERRAIN_SCALE_X2));

    // test if column height is greater than current voxel height
    // for current step from intial projection point
    if (column_height > z_ray)
    {
```

What Now?

Well you have a fast terrain generator with which you can alter terrain and texture data at will. I suggest that you try animating the height data, and/or the texture data. For example, you could create a tiled texture and height map and play animations on the data. You could animate real waves on the water, or little people running around the terrain. In addition, we've said nothing of lighting. You could create a light lookup table with 256 shades of each color by using a least-squares method. Then you could have another 2D light map that would describe the light intensity for each pixel in the texture map data. Then, as you're rendering, you could simply do one more look up per pixel to do lighting in real time. This would allow you to do lighting effects such as local spot lights just by painting into the light map each frame. Finally, there's no reason why you couldn't use a 3D polygon engine to place objects above the terrain. This way, you could make a poor man's FURY III. Have fun. ■

André LaMothe has written three best-selling game programming books and is currently CEO of Xtreme Games. He completed his latest project, REX BLADE, a DOOM-style game, in less than six months.There's not much left of him after that! You can reach him at necron@inow.com or http://www.rexblade.com.

LISTING 3 (CONT.). *The Terrain Rendering Algorithm.*

```
    // we know that we have intersected a voxel column, therefore
    // we must render it until we have drawn enough pixels on the
    // display such that their projection would be correct for the
    // height of this voxel column or until we have reached the top
    // of the screen

    // get the color for the voxel
    color = color_map_ptr[map_addr];

    // draw vertical column voxel
    while(1)
    {
    // draw a pixel
    *dest_column_ptr = color;

    // now we need to push the ray upward on z axis
    // so increment the slope
    dz+=dslope;

    // now translate the current z position of the ray by
    // the current voxel scale per unit
    z_ray+=curr_voxel_scale;

    // move up one video line
    dest_column_ptr-=SCREEN_WIDTH;

    // test if we are done with column
    if (++curr_row >= SCREEN_HEIGHT)
      {
      // force exit of outer stepping loop
      // cheezy, but better than GOTO!
      curr_step = MAX_STEPS;
      break;
      } // end if

    // test if we can break out of the loop
    if (z_ray > column_height) break;

    } // end while

    } // end if

    // update the position of the ray
    x_ray+=dx;
    y_ray+=dy;
    z_ray+=dz;

    // update the current voxel scale, remember each step out
    // means the scale increases by the delta scale
    curr_voxel_scale+=dslope;

    } // end for curr_step

    // advance video pointer to bottom of next column
    dest_buffer++;

    // advance to next angle
    raycast_ang--;

} // end for curr_col

} // end Render_Terrain
```

Back

LINEAR VELOCITY EXTRACTION FOR CHARACTER ANIMATION

I recently moved from the Lincoln Park neighborhood in Chicago, to Bucktown, another neighborhood within the city. Actually, the Salvation Army moved most of my furniture for me, so moving the balance of my possessions seemed, at first, like an easy task. I didn't bother calling a full roster of friends/movers for this move; I only called my friend Paul and his fiancée Sarah. The three of us moved all of my stuff, plus some of my own fiancée's stuff, in just under four hours. Even if it hadn't been one of the hottest days of the summer, there was just no way to make this process fun.

The moving process got me thinking. If I ever get a shot at a job in a video game publisher's marketing department, I'm sure I'll suggest game ideas constantly. I imagine myself sitting behind a big desk with brand

B Y S C O T T C O R L E Y

new business cards and plenty of time on my hands. I'll have a big development staff somewhere close by, and I'll be sure to dismiss all of their ideas as unmarketable. Then, one day, when I have a particularly open schedule, I'll drop the big one. I'll insist we develop a "moving day" game in 3D. I'll call it MOVING DAY 2000 X-3D.

Using my MOVING DAY 2000 X-3D game as an example, I'll explain a concept called linear velocity extraction (LVE), a technique for managing the animation of walking creatures. The techniques that I describe could be used to create a homespun development tool for modifying and managing these animations. The August 1997 issue of *Game Developer* contained an article called "Using a Base-Root System for Animated Characters," which is heavily related to this topic. I'll re-address some of the concepts introduced in that article to provide some context, but if anything seems brushed over or confusing here, refer back to that article and read it.

How to Play MOVING DAY 2000 X-3D

The object of MOVING DAY 2000 X-3D is to move all of the furniture, boxes, and junk from one dwelling to another in four hours or less. If you succeed in moving everything in the allotted time without excessive breakage, your buddy buys you lunch. If you exceed the time limit, U-Haul charges you double for the moving van and hassles you about refunding your $120 deposit. If you damage too many of his belongings during the move, your buddy smacks you.

When the game starts, you get to choose one of four characters with which to move your friend's boxes. All of the selectable characters have two arms and two legs, and they all use their legs to get around. Once you select a character, your buddy tells you which boxes need to be moved, which boxes are particularly fragile, and how much time is left. For easy reference, we'll call your in-game character "Sucker."

Sucker is a bipedal creature in a real-world environment with gravity, so most of his movement will be confined to a 2D plane. Aside from jumps and special moves, the motions we capture will fall into the categories of standing, walking, jogging, lifting, carrying, dropping, throwing, and setting down nicely. Within these categories, we may have a wide variety of interchangeable animations to mix things up a bit. For lifting, carrying, throwing, and setting down, we'll choose an animation based on the weight of the box involved. The final motion capture list will have around 150 moves on it.

We would like to maintain appropriate velocities for each animation. Back in the days of sprite animation, it was important to use an appropriate velocity for a moving animation so that the sprite's feet would not "slip" on the ground. In 3D, using an appropriate velocity is still important, and in some cases close-up camera views make foot slippage even easier to detect.

Fortunately, the translation involved in a 3D animation is already stored as pure numbers, so determining the velocity of an animation is easy. It's certainly easy compared with determining the velocity of a sprite animation.

Allow me to insert some standards here. I follow the same concepts used in my August article, so if you read that, you might want to skip this paragraph. Our floor (the plane that Sucker walks on) is the plane z=0. The world is a right-handed coordinate system where the positive z axis is "up." Sucker has a root node that is at the top of his skeleton hierarchy; move the root and Sucker moves in the world. This root node is positioned at Sucker's center of gravity (the joint between the torso and the hips). This root node is the only part of the animation that will have changing

Crafting a Tool to Process Character Animation Data

During the course of this article, I mention this mysterious "tool" that you'll need to write. Most of you have tools to deal with 3D characters and animation data already, and can modify them to implement new features. If you don't have any tools to work with the files spit out by your animation package, you need to write some, so read on.

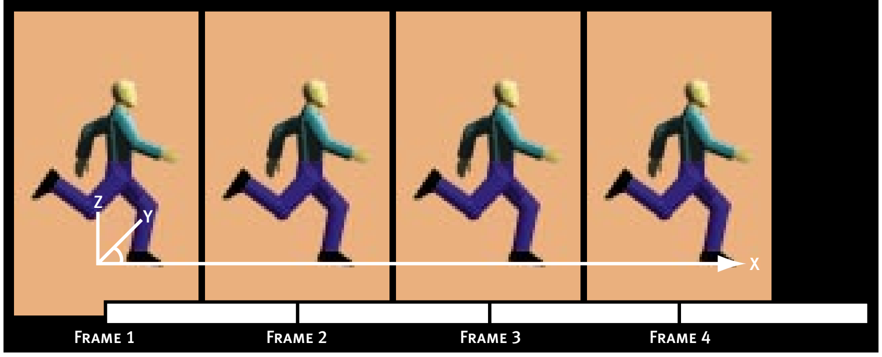
The tools that you write should accomplish a number of important things for you. First, most commercial 3D packages have fairly complex file formats that store a ton of information that you don't need during your game. Your tool should extract only the information of interest to you. Second, there may be a native format that you need your data in; for example, a fixed-point format. Your tool should handle that as well.

Getting information about converting 3D file formats varies from easy (look in the back of the instruction manual) to almost impossible and expensive. Before considering a 3D animation tool, you may want to investigate how easy it is to extract data from the file formats. I won't say specifically which vendor I've had the most problems with, as that would be très gauche, monsieur. I will remain soft, so as not to spoil anyone's image.

Most 3D tool vendors supply linkable libraries to extract the data from their files. This is convenient because if the file format changes, all you need to do is link new libraries, and you're ready to go. I'd be wary of this approach for a few reasons. First, the libraries may not give you some bits of information that you need, and since you can't modify the libraries, you might be out of luck from the start. Second, the libraries may be provided in a form that isn't compatible with your linker or your CPU. Third, that bit about linking with updated libraries when new file versions come out is a bit flawed. Many 3D tool vendors are slow to get new developer tools released, so if a small change in the file format makes the libraries useless, you have no recourse except to wait for new libraries. If you write your own tools to parse through the files yourself, you can at least attempt to handle new file formats before all the information becomes available to you (and usually this is very easy to do).

So go write a behemoth tool that can read in 3D hierarchical animated characters, process them to the point that they are useful to you, and spit them out in a format that is convenient for your purposes. You'll be glad you did.

FIGURE 1. *As animation progresses, the root moves away from the origin.*



translations in x, y, and z, in addition to rotations. Other segments will have translations from their parent, but the translations will stay constant for all animation frames. You may be wondering why I use z=0 as the floor instead of the popular y=0 floor. Some professional 3D tools use z=0 (notably, MultiGen's tools), while others use y=0. I prefer z=0 because it puts all movements on the floor in terms of x and y. If all graphics tools agreed on one standard, I'd use it. As long as the tools you write provide a means of rotating the axes to a common orientation, the standard you use doesn't matter.

Sucker's root will always be animated so that his feet align properly with the z=0 floor. We'd like to control Sucker by using a base coordinate system projected onto the floor. On the programming end of things, it's easy to create this base coordinate system, because everything in our animation is relative to the origin. As long as the base coordinate system is located at the origin, everything Sucker does will be relative to that base, and we can move Sucker relative to the world by changing the base's translation. The only rotation we'll apply to the base will be around the base's z axis, so you can envision the base as being an imaginary disc that is always parallel to the floor. In addition, the base's z translation will always be zero, so that imaginary disc will actually be incident with the floor plane.

The Rule of Base

So far, we've covered the basics from the original base-root system article — we can now get into a discussion of LVE. If you followed the preceding quick overview of the base coordinate system, you should now be interested in the details of what to do when Sucker translates away from the origin. We know that the base coordinate system is always located at the origin of the space that Sucker was modeled in, but we also know that Sucker can walk away from the origin in that space, as in Figure 1.

If Sucker's root isn't moving a significant distance from the origin, we don't need to do anything. If Sucker is just standing around, or juggling while standing, or even doing jumping jacks to warm up, no adjustments need to be made. During any of these animations, the root may move around quite a bit, but at the last frame of the animation, the root won't be very far from the base. The "rule of base" says that the base can always be considered an approximate projection of the root onto the floor. As long as this remains true, we're fine.

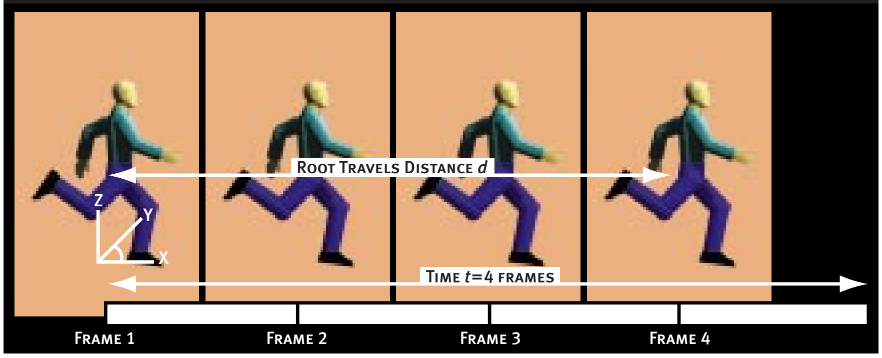
When Sucker walks or runs, the root can be a few feet from the origin at the end of the animation. This violates the rule of base, so an adjustment needs to be made. That adjustment can be determined using LVE.

The idea behind LVE is very simple. Our animated character moves a certain distance d over a certain time t. We can calculate d simply by taking the final position of the root and subtracting the initial position of the root, as shown in Figure 2. Determining t is simply a matter of reading the total number of frames or the total animation time from our animation file. Note that I gratuitously use the word "simply" here. If you don't have the tools to read and process your model and animation data directly, you'll have to write those tools before you can do any of these calculations. That may or may not be a simple job. Once you have those tools running, though, you can reread this paragraph and truly enjoy the word "simply."

Now that you have d and t, you can calculate v, the linear velocity, as d/t. Actually, d has three potential values: d_x, d_y, and d_z. For now, we'll leave everything in z untouched, so forget about d_z. Calculate d_x (final x position minus initial x position) and d_y (final y position minus initial y position). Then calculate v_x and v_y, the x and y linear velocity components, by dividing d_x and d_y by t. The final units you end up with might be feet per second, feet per frame, meters per second, or undefined distance units per unknown time unit. The units don't matter, just make sure you know what they are. You and the animators giving you this data need to know what 1 equals. You can get pretty far without ever knowing this, but eventually you'll need to know.

Once we have the linear velocity, we can put the rule of base back into effect. Before we do that, however, I'd like to explain the overuse of the word "linear" in this article. The word is used to emphasize the fact that we're interested in linear distance change

FIGURE 2. *Calculate velocity of animation using distance traveled and the overall time of the animation.*



The Rule of Base

The Rule of Base states that the base coordinate system may always be considered an approximate projection of the root coordinate system onto the floor. Whenever this rule is violated, you must take special action. In almost all cases, LVE is the best way to restore the Rule of Base.

over time, not angular velocity or anything else. It's also used to emphasize that the actual movement of the root of an animated character is very erratic. The root is translating through all sorts of arcs, shaking around, and at times, may even be moving backwards relative to the ground when the animation as a whole has a forward velocity. The velocity that we're extracting is a simple distance over time. The rest of the wacky movement is left in the animation. When we add the two back together, we reconstruct the original motion. We're almost there.

To put the rule of base back into effect, we have to adjust each frame of the animation by subtracting a certain distance from the x and y translation from each frame. Use Equation 1 to calculate that distance for a particular frame. In Equation 1, we'll use the two linear velocity components v_x and v_y, the time stamp f of the frame, the original components of the frame's root translation x and y, and the frame's new root translation components x′ and y′.

$$x' = x - f*v_x$$
$$y' = y - f*v_y \qquad \text{(Eq. 1)}$$

The time and distance units in Equation 1 are up to you. Again, whether you're using feet per frame or meters per millisecond, you need to define your units and remain consistent. Check out Listing 1 for an example of how this is done in code.

So what have we done? Equation 1 is simple, but it was difficult to explain, so it must have some significance. For each frame in the animation, Equation 1 subtracts the distance that the root has traveled up to that frame. If you play back a run animation after doing this LVE processing, the animation will appear to run in place. We have put the rule of base back in effect. When saving out the processed animation, store v_x and v_y with it, as we'll be using that extracted velocity very shortly.

Books Are Heavy

We can now play back an animation of Sucker carrying a large box full of books. If you just play back the LVE-processed animation, you know that the animation won't move away from the base (the rule of base is still in effect). To get this animation moving without violating the rule of base, you have to move the base. You stored the extracted linear velocity with the motion, so you know exactly how fast to move the base.

The animation system that you write for MOVING DAY 2000 X-3D should automatically read the velocity for each animation and apply it for you, so you never have to adjust any of the velocities by hand. In addition, any funky root movement from the original animation has been retained. The extracted linear velocity can handle any forward or backward movements along the x axis (we assume Sucker is animated facing down the positive x axis) as well as any sideways strafing movements along the y axis. In fact, it can handle any movement in the x-y plane.

Recall that at the beginning of this article, we decided to leave the z movement intact in the animation. This turns out to be a good decision in light of the "linear" emphasis. Sucker lives in a world with the acceleration of gravity, so motion in the z direction certainly isn't going to be linear. For this reason, we don't do any extracting in the z axis.

If you refer back to the juggling and jumping jacks paragraph, you'll recall that there will be animations in which the rule of base is followed without any processing. For these animations, we don't want to extract a velocity at all because it will make Sucker drift when he's supposed to be standing still. There will almost always be a difference in translation between the first and last frames of an animation, but if that distance is small, we would like to leave the animation alone and store a velocity of zero. The question then becomes, "What's a small distance?"

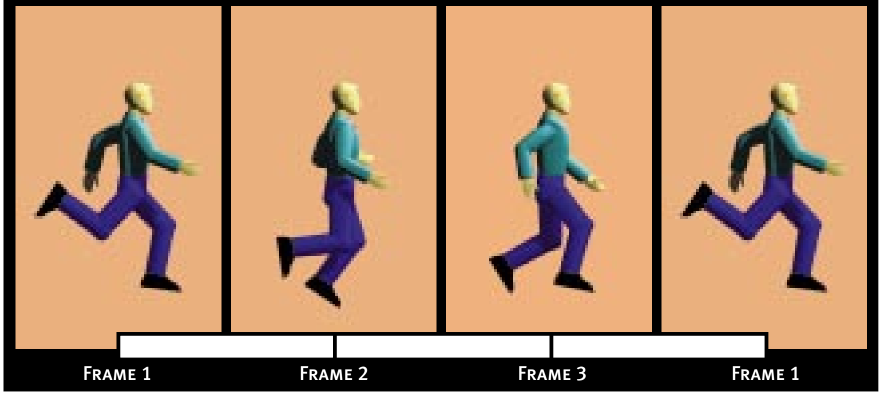
What you consider a small distance is up to you and can be handled many different ways. A perfectly reasonable method of determining "small" is passing in a distance on the command line of your tool. Another method to consider is examining the animated model and using some measurement from

LISTING 1.

```
// Note: This code snippet doesn't cover the definition of the classes
// it uses, such as Animation and Model. I will explicitly use the "this"
// to reference class attributes, for clarity. Also, you
// can assume that no global variables are being referenced in this code.

void Animation::TakeLinearVelocity(Model *theModel, double &xvOut, double &yvOut, double
     &zvOut, bool ForceVelocity)
{
     double modelSize[3];
     double velocity[3];
     double start[3],end[3];
     double curPos[3], dist[3];
     int i,j;

     theModel->GetMaximumDimensions(modelSize[0],modelSize[1],modelSize[2]);
     // determine what "small" is

     modelSize[0]/=4;          // If animation moves the root a distance greater than
     modelSize[1]/=4;          // 1/4 of the largest dimension of the model,
     modelSize[2]/=4;          // we will extract a velocity.
```

Conitnued on page 51.

FIGURE 3. *Animation assumes last frame is identical, in time and position, to the first frame*



that to determine "small." For example, if you have a segmented model, you might find the largest dimension of the largest segment and declare that to be "small." This would mean that no velocity would be extracted unless the root moved a distance that was large relative to the model itself.

Once you have a value for "small," compare your d_x and d_y values to it. If a distance in a particular direction is greater than "small," extract a velocity in that direction. If not, set the velocity for that direction to zero and leave the translation in that direction alone.

By this time, you should understand the importance of deciding what your units are and keeping them consistent. There's another sticky consistency problem that you might not notice, which can make your animation loops look horrible. You and your animators need to define where the first and last frames of your animation are, and how they will be handled. Also, verify that this definition of the first and last frames works with the animator's software.

Handling Looped Animations

The first frame of a looped animation is obviously where the animation starts. It's time zero. While frame one is obviously time zero, it very well may be considered frame one by the animation software (only programmers count from zero). This shouldn't cause a problem, but is something to keep in mind.

The last frame of a looped animation is, of course, where the animation ends. You can insist that the animator make the first and last frames of the animation identical, in which case the first and last frames may be considered identical in time (at the loop point, the last frame is the first frame and occurs at time zero). In other words, there would be no time between the last frame and the first frame, as in Figure 3. When you play the animation back in MOVING DAY 2000 X-3D, you'll play through the animation and loop back to the first frame, skipping the last frame, because it's identical to the first frame in appearance and in time.

There are a few problems with the above situation. If your tools aren't written carefully, they may include the first and last frame in your data. Since these frames are identical, and we're really skipping the last frame, including both frames would be wasteful. The second problem is that your animation tools may not work this way.

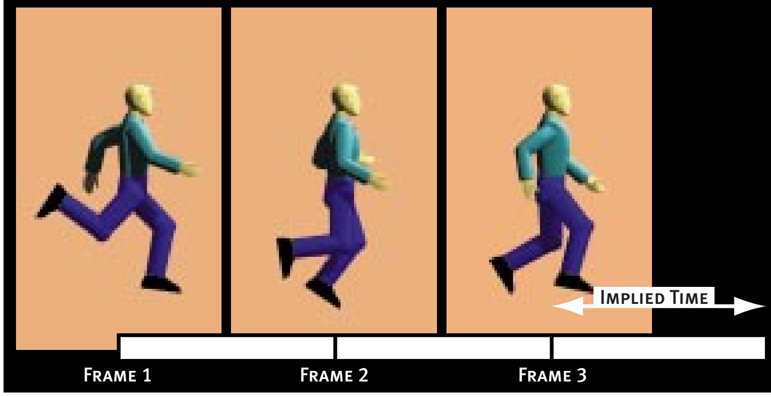
The straightforward way of dealing with the last frame of a looping animation is to assume that there is one implied frame of time between the last frame and the first frame, as in Figure 4. In other words, the last frame should be the frame that comes just before the first frame, so when you loop the animation, everything looks smooth. There's no need for your tools to leave off a frame, because there's no redundant frame at the end of the animation. The problem with this method is that again your animation tools may not work this way. Either of the loop methods that I just presented is acceptable, and if you reread those sections you'll see that the methods are essentially identical.

Speed Shifts

Now that Sucker has his automatically ripped-out linear velocities to propel him, you may want to adjust the velocities at game time. When the end of the day nears and Sucker gets tired, you may want him to move more slowly than usual.

When you adjust a velocity, start out with the original velocity vector for the animation that you're adjusting. (Up until this point, I've referred to the velocity by its components v_x and v_y. The velocity vector that I refer to here is made up of these components.) Multiply the x and y components of that velocity vector by a scale value. For example, you could multiply by 0.8 to slow Sucker down 20%. The only other thing you need to do is multiply the playback frame rate of that animation by 0.8 as well so that Sucker's feet won't slide. While this is easy, when creating animation system tools you may not initially think to provide a way to override an

FIGURE 4. *Animation assumes that there is time between the last frame and the first frame.*



LISTING 1 (CONT.).

```
    // If ForceVelocity==true, we will extract a velocity no matter how
    // small the overall root translation is. A good way to force this is
    // by setting our "small distance" value to 0.
    if (ForceVelocity)
      modelSize[0]=modelSize[1]=modelSize[2]=0;

    // When an Animation is constructed, its velocity is initialized to zero.
    // If we detect that the velocity is not zero, it means we have already
    // extracted a velocity. Repeating this step again would be disastrous,
    // so here we do a safety check.

    if ((this->xVelocity!=0.0)||(this->yVelocity!=0.0)||(this->zVelocity!=0.0))
    {
      xvOut=this->xVelocity; yvOut=this->yVelocity; zvOut=this->zVelocity;
      return;  // Velocity was already extracted. Don't repeat!
    }

    velocity[0]=velocity[1]=velocity[2]=0.0;      // Get ready to find the velocity
    if (ForceVelocity)
    {
      // In the case of a forced velocity, the velocity being forced is passed in
      // via xvOut, yvOut, zvOut
      velocity[0]=xvOut; velocity[1]=yvOut; velocity[2]=zvOut;
    }


  this->GetFirstRootTranslationFrame()->GetPosition(start[0],start[1],start[2]);
  this->GetLastRootTranslationFrame()->GetPosition(end[0],end[1],end[2]);

  // Determine linear velocity and subtract it from each keyframe
  for (i=0; i<2; i++)
  {
      if (abs(end[i]-start[i])>=modelSize[i])
      {
        dist[i]=end[i]-start[i];
        if (!ForceVelocity)
          velocity[i]=dist[i]/(this->LastFrameNumber - this->FirstFrameNumber);
    // This divisor represents the number of >spaces< (of time) between keyframes

        for (j=0; j<=(this->LastFrameNumber - this->FirstFrameNumber); j++)
        {
          this->GetFrame(this->FirstFrameNumber + j)->GetTranslation(curPos[0], curPos[1],
                curPos[2]);
          curPos[i]-=velocity[i]*j;
    // Subtract the distance that the velocity has taken us on frame j
          this->GetFrame(this->FirstFrameNumber + j)->SetTranslation(curPos[0], curPos[1],
                curPos[2]);
        }
      }
    }

    xVelocity=velocity[0];
    yVelocity=velocity[1];
    zVelocity=velocity[2];
    xvOut=xVelocity;
    yvOut=yVelocity;
    zvOut=zVelocity;

}
```

animation's velocity and frame rate.

If Sucker seems too responsive to the joystick for some reason, you may have to give Sucker a slight acceleration, from zero velocity up to the velocity of a walk or run animation. This is achieved in the same way as adjusting an animation's velocity, except in this case the velocity is changing every frame, so the playback frame rate must be adjusted every frame as well, until Sucker accelerates up to his target velocity.

Tips for Tuning Animations

Here are some tips to polish up your animation tool. Add command line options to your tool that will suppress LVE completely (in both the x and y axes), or partially (either x or y only). When deciding your value for "small," you'll probably find that no value is perfect. You may get an animation that has just enough root movement to get a velocity extracted, and it may cause your character to drift when you want it to stay still. In addition, you may find that some animations get a y velocity extracted when they should really only have a x velocity.

Strive to adjust your "small" value so that the vast majority of animations convert automatically with no special intervention. This will reduce your workload and data maintenance problems. When certain animations give you trouble, just make sure that they're converted with the appropriate LVE suppression flags.

A particularly good candidate for LVE suppression is any animation in which a character pivots on one foot. In these cases, your root is going to swing 180 degrees and, in the process, travel a significant distance. However, this special case requires a solution using some method other than LVE. You definitely don't want a velocity extracted from an animation such as this, so use those LVE suppression flags.

Provide a way within your animation tool to force the extraction of a specified velocity from an animation. Normally, your tool will determine the velocity of each individual animation and extract that velocity. There will be cases where you want to override the velocity that your tool calculates with a specific velocity from elsewhere. Sucker provides a perfect example of when we might want to do this. There are objects in Sucker's environment that he interacts with, and we will have animations associated with those objects that match up with Sucker's animations. For example, Sucker will at times pick up boxes, carry boxes, throw boxes, and set down boxes. The motion of the boxes in these cases has to be animated and synchronized, since we don't want to simulate the box as a rigid body resting on Sucker's arms.

During the animation of Sucker and the boxes, you'll have to make sure that all of the animations are of the same length and that they look correct when played together (that much is obvious). What isn't clear right away is that when you process the animations separately, you'll get slightly different velocities for Sucker and the box due to slight differences in the starting and ending positions of the roots. If you then play back the animation with Sucker and the box synchronized perfectly, the box and Sucker will eventually drift apart. We must therefore come up with a way to match their velocities.

For all animations involving Sucker and a box, we'll convert the two animations as a pair and use Sucker's velocity for both. First, we convert Sucker and make a note of the velocity vector that we extracted. Then, we use our tool's velocity override feature and force the box animation to have the same velocity as Sucker. Now, we can play the two animations in sync and not worry about them drifting apart.

The last point I'll make about synchronized animations is that if they are animated together and then separated later, their base coordinate systems will match exactly. When playing the animations back in the game, the bases should start out in the same place with the same facing. If the paired animations' velocities match correctly, the bases will stay matched up, which is handy.

Go Forth and LVE

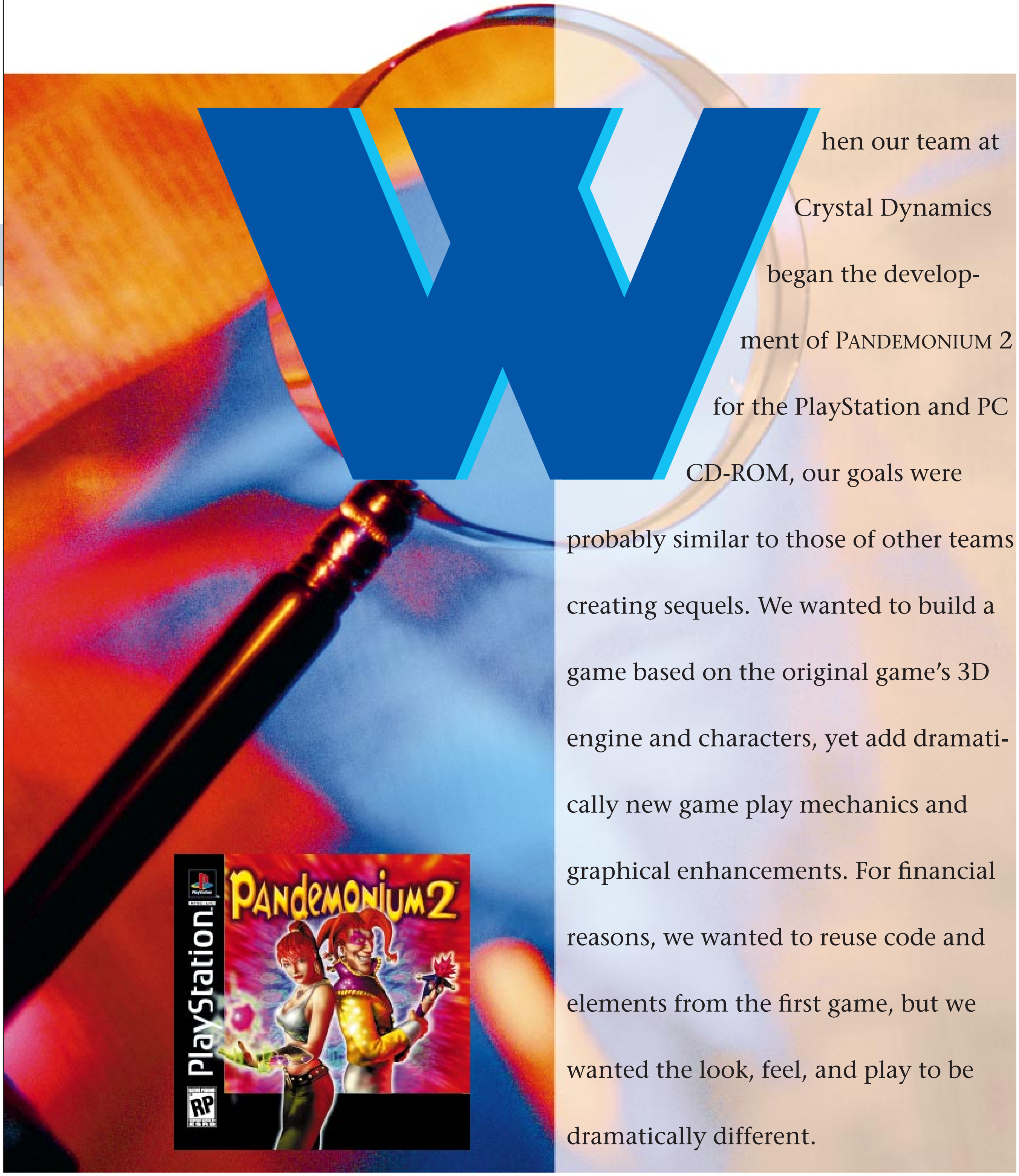
That wraps up our discussion on LVE. Depending on your experience creating and managing animations, this may seem like a lot of work. If it seems like a lot of work to create a home-grown tool to manage LVE, be aware that for many games on the market today, approximately half of the coding effort was spent on creating tools alone. From that perspective, implementing LVE should be a fairly insignificant blip on your workload.

If this doesn't seem like a lot of work, try it out. It's fun to have tools that do everything automatically for you. By the way, after this investigation of MOVING DAY 2000 X-3D, I have decided to scrap the project and reallocate the resources elsewhere. See ya, Sucker. ■

Scott Corley is vice president of software development at High Voltage Software. His new wife actually thinks game programming is cool. For tips on where to find such enlightened women, e-mail scottcy@ripco.com.

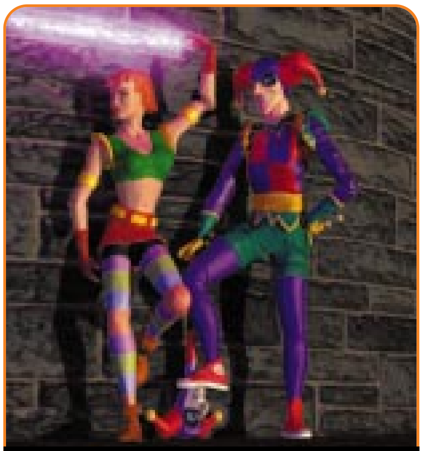
Back

Pa demo i m 2



When our team at Crystal Dynamics began the development of PANDEMONIUM 2 for the PlayStation and PC CD-ROM, our goals were probably similar to those of other teams creating sequels. We wanted to build a game based on the original game's 3D engine and characters, yet add dramatically new game play mechanics and graphical enhancements. For financial reasons, we wanted to reuse code and elements from the first game, but we wanted the look, feel, and play to be dramatically different.

The PANDEMONIUM experience was built to mimic a fast-moving roller coaster. The engine in the original game, which shipped in October of 1996 for the PlayStation, supported a very active camera that delivered cinematic angles and a wild sense of motion. We wanted to build on that foundation and design an even bolder camera scheme. We also wanted to communicate the sense that the game was "hand-crafted," because our development team feared that originality was being lost in the character-action genre, and that publishers were continuing to reuse a significant amount of tile sets and backgrounds to the detriment of creativity. Fooling the consumer with empty claims of large numbers of


Nikki, Fargus, and Sid in the original PANDEMONIUM...

...and Nikki's updated look for PANDEMONIUM 2.

PANDEMONIUM DOESN'T JUST DESCRIBE THE FRENETIC PACE OF DEVELOPMENT AT CRYSTAL DYNAMICS, IT ALSO REPRESENTS THE COMPANY'S BREAD AND BUTTER THIS CHRISTMAS

by Scott Steinberg

levels, when the number of *unique* levels remained somewhat modest, grated on our gaming sensibilities. Our interests were immediately focused on building originality and creativity throughout the entire product.

With regards to graphics, we wanted to push into a new art frontier with a thematic look that was radically different from one level to the next. We also wanted to aggressively augment the dynamic roller-coaster camera and kinetic horizontal game play, while at the same time build the same spatial drama with vertical game play.

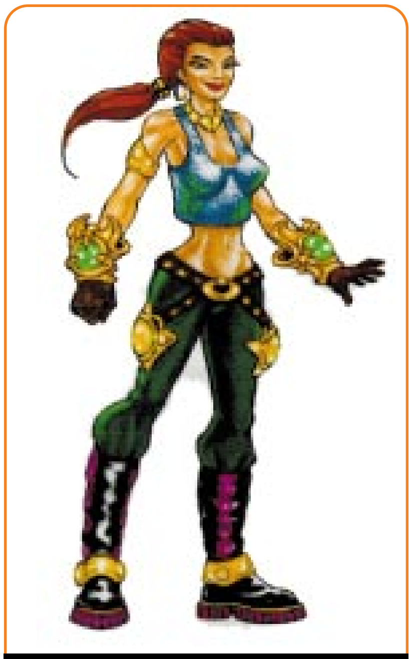
Before we could start development, however, our team had to be relocated from our studio in Novato, California, to our main office in Menlo Park, where the majority of Crystal Dynamics' staff is located. While only two hours away by car, the new location was a significant change in the team's working environment and caused some friction between team members, as I will explain shortly.

The Genetic Core

One of the core strengths of the PANDEMONIUM development team stemmed from the empowerment of individual designers to dream up and execute graphics or game play ideas without hindrance from the other development disciplines. Our development environments were custom-built with powerful tools to allow designers, artists, and programmers to experiment, innovate, test, and build new features before they were hard-coded into the game. These tools took about three to six months for a person to learn. Every team member had a Pentium 166MHz with a 3D accelerator card and Kinetix's 3D Studio 4.0 — essentially all workstations purchased off the shelf without much further tweaking. Our commitment to individual creativity and flexibility during the development of the original product allowed the team to quickly transition to the sequel without our having to retrain them or invest heavily in new development resources. In addition, by strange coincidence, several members of the design team were high-school friends who were unknowingly hired together, which helped fuse the group.

Another one of our goals was to surpass the graphical standards set by previous games. From the outset, there was little interest in repeating the original PANDEMONIUM's fantasy environments. Instead, our team decided to push in a direction reminiscent of Salvador Dali and Timothy Leary, which sets PANDEMONIUM 2 apart as a visual wonderland. This aesthetic goal was balanced by a core game play directive: to build a layered and integrated entertainment experience based on technical — as well as exploratory — game play.

The sound design of PANDEMONIUM 2 was a subtle but extremely important feature to the development effort. From the beginning, the music designers adopted a direction radically different from the rest of the industry. Character action games from the U.S. and Japan historically have used fantasy-oriented music. With the capabilities of the CD at their fingertips, the designers chose a new genre for inspiration: the San Francisco club scene. Their influences came from a host of esoteric musical sources such as Freaky Chakra, Tipsy, Steroid Maximus, Sukiya, and Bill Laswell. Exploring various clubs in the city, the sound team crafted a sophisticated and incredibly energetic musical skeleton. This musical direction excited our international distribution partners, as it was



Nikki, the heroine, was completely redesigned for PANDEMONIUM 2.

extremely relevant to their consumers and offered multiple marketing opportunities and partnerships.

Character Design

The most significant change planned for the sequel involved retooling the two main characters to give them a more mature look and attitude. Nikki, the heroine, was completely redesigned. Fargus, the goofball jester, was given enhanced mechanics and a more demented look and behavior. Multiple looks for the characters were presented to focus groups to gauge consumer appeal. In turn, we ran these models by our distribution partners in Europe for review and buy-in. This level of involvement from our UK, French, and German distributors was key to maintaining European consumer appeal with this sequel. The process was iterative; it took multiple meetings and character sketches before a satisfactory look and feel was achieved. While some people might have built a sexy character à la TOMB RAIDER's Lara Croft, thankfully nobody in the company was interested in cloning Croft. Instead, we went after a more elegant look, somewhat reminiscent of Elle MacPherson. Agreeing on this artistic target helped galvanize the relationship between the development team and the marketing team. Both groups made constructive contributions to the creative process and to the final result.

Once the rough designs were finalized, the characters went into their 3D modeling phase. Lead artist Leon Cannon and lead designer Zak Krefting bore much of the pressure to complete this task in time to preview the game at 1997's E3 in Atlanta, and to meet marketing deadlines for advertising and packaging. While Nikki's character was somewhat challenging to complete, Fargus was extremely time-consuming to finish. His costume went through many iterations, many of which made him look like a psychotic harlequin jester. While it was an interesting and funny look, it wasn't quite the treatment we had all envisioned. Eventually, his appearance evolved to meet the tastes of the team. Fargus' partner Sid, a puppet on a stick, was designed to be his alter-ego throughout the sequel. Sid's role in the game was greatly expanded as a range weapon, as well as game play mechanic.

Fargus and his sidekick Sid were given a more demented look

The Quest for Beta

Achieving a feature-complete beta proved to be a challenge. Our release date was moved up by almost a month for marketing reasons — we viewed November and December as a crowded time for game releases. As a result, our development schedule was compressed The team had a hard target date in August 1997 that forced everyone to work concurrent days without sleep. CDs were burned and distributed for game play feedback and testing. Compressing the schedule meant that certain key elements of the game weren't completed until the last phase of development.

Shortening a product development schedule by nearly a month is unprecedented in our industry. It requires a massive amount of work, planning,

The talented yet ethereal Pandemonium 2 development team, hard at work fostering a creative work environment.

and coordination. In order to achieve our new ship date, some of the original Pandemonium team members, who had been working on another project, had to be brought in to help complete the sequel. These four "volunteers" signed on for a five-week tour that made a huge difference in our ability to make the early ship date. Once their tour of duty was over, our team was left to finish up the schedule and ship in October. Although we will meet our October ship date for the PlayStation game, the PC version will be delayed until later in the year.

Four weeks before our golden master date, the team struggled to update glue screens, complete the introduction's sound mix, and stabilize numerous crash bugs. Two weeks before the golden master date, the bug list was still significant, and many felt that the game was much too difficult to play. The marketing department sent reports to our team noting where consumers were encountering difficulties, and we frequently turned around new versions of the game that reflected this real-time feedback.

Compounding our technical challenges, we had to overcome some internal misconceptions about the project. Since Pandemonium 2 had been positioned internally as a sequel, many people assumed that the game represented nothing more than a modest technical challenge. On the contrary, the project was quite challenging. The functionality of the new characters' features and behaviors, along with enhanced enemy intelligence, made the product quite a technical undertaking.

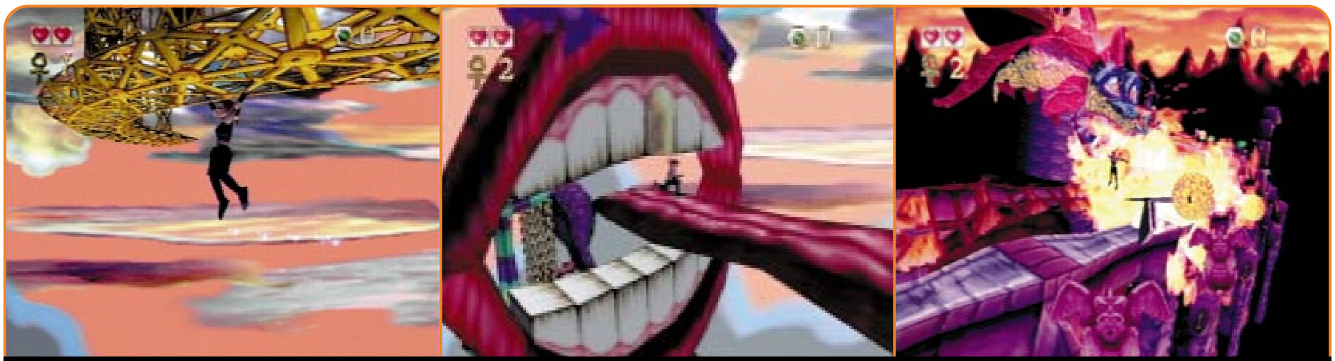
Lead designer Zak Krefting.

The Gauntlet

The approval cycle for a PlayStation product is very stressful. With in-store commitments and marketing activities timed to coincide with the retail availability of a product, all eyes focus on the testing cycle. Costly delays can arise if the tenacious staff within Sony's quality assurance department reject a product due to bugs in the software. Our team felt that our biggest potential bug risks stemmed from Pandemonium 2's new character features. These fears proved to be well founded, as early on in the testing process the testing team discovered a disproportionate number of glitches associated with the new character features.

Communication between testers and the development team was especially challenging. In order to provide useful and constructive feedback to the development team, the Pandemonium 2 testers required special training from the test department. For instance, the Pandemonium 2 camera movements have their own lexicon, which required the design and testing staffs to identify, itemize, and diagnose camera and design bugs using a very technical and specialized vocabulary. Thankfully, many of the original Pandemonium testers were available to test the sequel, which

Lead artist Leon Cannon.


PANDEMONIUM 2 *features a dynamic, roller-coaster camera and kinetic horizontal game play.*

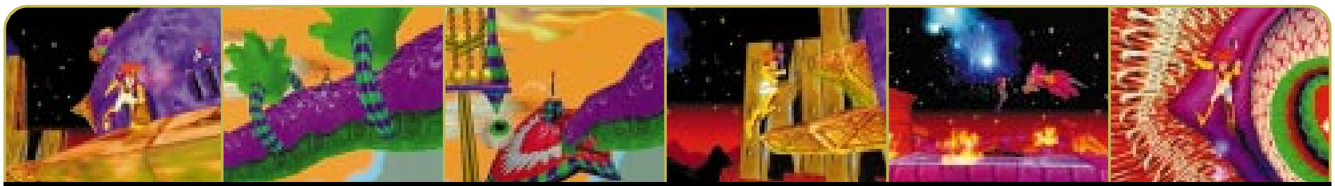
helped minimize the department's learning curve.

What Went Right

1. From the outset, the team was driven to design and create a large number of very diverse levels. This goal was partially met on the design side and truly met on the graphics side. When the decision was made to shorten the development schedule and ship the product in October, design features that we had planned to include beyond the core mechanics were abandoned. However, the product was graphically rich and diverse from the outset, so the shortened schedule didn't have the same impact on art that it did on the game design.
2. From the beginning, our development group established an unconventional management, art, and design philosophy. Given so much individual autonomy, our group had a magical feeling to it, a sense that both art and design could fuse together and integrate like never before. This integration allowed us to blur the traditional lines of team hierarchy and development specialties. On a more subtle level, our structure also allowed individual personality traits to shine through in the game play and level layout. Core team members exhibited a wide spectrum of game play styles and wove their personalities into the fabric of the game.
3. There was an almost religious fervor within our team, riding the power of the development environment, the game's engine, and the group's creative cadence. Team connectivity, integration, and involvement in the creative process was felt by all to be the key to success. The more new technology tricks and the more brains contributing to the game design, the better the product stood to be.
4. A by-product of the design team's remote location during development of the original PANDEMONIUM was the team's general distrust of upper management and the marketing group. Without personal contact on a daily basis with these groups, the design team had demonized marketing and management. The situation improved significantly during development of the sequel, when everyone worked in our Menlo Park offices. There was a collective sense of involvement shared by the executive staff, marketing, and the development groups, and much of the tension (particularly between the development team and the marketing team) quickly evaporated. Some of the techniques used to facilitate this spirit were fairly straightforward and occurred organically. They centered on getting people to work together at consumer focus groups, reviewing character sketches and designs, encouraging involvement, and valuing each others' opinions based on our various areas of expertise.

   There were two specific situations where both the marketing and development teams came together for a common goal. The first was when the marketing team orchestrated a media tour that yielded a potential magazine cover in a leading video game magazine. To secure our spot on the cover, marketing needed to create stunning artwork that would impress the magazine's editorial decision makers. With this challenge in hand, the marketing and art teams worked together over the course of several days to create and polish a suitable scene. The result was that the magazine's editor was ecstatic about the treatment, and we landed the cover.

   The other instance that tested our teamwork occurred when packaging artwork arrived from an out-of-house design team. The character models in the artwork were deemed unacceptable, so with time in short supply, the marketing team again solicited the artists to enhance the artwork. Again we met marketing's schedule and quality expectations.
5. We created "micro teams," built around designers and artists with a similar vision. This reengineering of the traditional team brought along with it a renaissance of creative freedom and efficiency. The micro teams were built to guide the creation of each "game zone," or chapter of the game. Decisions on game play, specific mechanics, and the graphical style were made within these micro teams. Of course, overall continuity was still extremely important, and was managed by the lead designer and the lead artist. To reinforce the overall leadership of the project, collective team meetings were frequently held to communicate each micro team's zone specifications. This helped minimize areas of redundant game play and let us share innovative ideas with the rest of the product team.



PANDEMONIUM 2's whimsical graphics were designed to be reminiscent of Salvador Dali and Timothy Leary

What Went Wrong

1. One of our initial goals was to address a major criticism of the original product: that the game play was too linear. Achieving this lofty goal remained elusive. Programming resources proved scarce during the start-up phase of the project, and during early diagnostic meetings, we realized the complexity of this goal and decided that our technical team was not big enough to reach this goal in time. Instead, we settled upon a more achievable target: to develop a system of multiple paths in various levels that branched off and let the player explore more of the game's environments.
2. Looking back at the team's dynamics, one of the loftier goals that we narrowly missed was our attempt to maintain the organizational freedom that the team enjoyed during development of the original PANDEMONIUM. That atmosphere fostered a spirit that bound the group together. Once the team consolidated in Menlo Park with the rest of the company, some of the designers felt that their frontier spirit would inevitably be lost. Little actions were spontaneously taken to revitalize that attitude, such when some team members set up a bright purple tent around a central work area. Yet some corporate organizational elements crept into their work lives, including endless demonstrations to distributor candidates and countless marketing meetings.
3. The team had planned to use in-game character models in animated sequences, rather than large, fully-animated sequences. Using character models would have been more affordable, easier to implement, and less time consuming than the fully animated sequences. Unfortunately, implementing character models wasn't as easy or as fast as we had originally thought. Since it wasn't on the critical path to shipping the game it was pushed back in the schedule. When our schedule adjustment was made, those animated sequences didn't have the programming or art support to be implemented.
4. In general, working on a sequel can be somewhat stifling and difficult to get excited about. As I explained earlier, one of the goals was to augment the linear game play of the original PANDEMONIUM, which brought with it many interesting creative challenges. In the early part of 1997, it became clear that the team would realize only a partial victory. This design augmentation was the primary motivation for many members of the team, so when we retreated from this target, the creative energy and morale of the team dropped. Managing expectations and creative energies turned out to be critical in keeping up enthusiasm and passion.
5. There was some controversy over whether to do a computer-rendered 3D introduction to the game. The marketing department was interested in developing a high-quality, multiscene introduction to help relaunch the characters and enhance the product's publicity efforts. The team was suspect of the value of the sequence when measured against the cost and time requirements to complete it. The art group didn't have the time, nor the inclination, to work on it, and it was farmed out to a freelance contractor. There were problems from the outset, causing major budget overruns and delays. Specifically, character motions and aesthetic integrity were continually problematic, to the degree that the project barely made it into the game by the beta testing stage. This late completion precluded taking advantage of prelaunch marketing and sales opportunities that arose after our alpha version was completed.

The Legacy

Taking into account all that's been said thus far, everyone associated with PANDEMONIUM 2 was quite proud of how it turned out. In this age of beefy product development schedules and budgets, the game did what few have ever done: It shipped early. The single most important lesson we learned was how important the human component is in development. Whether the term is "intellectual capital" or "talent," products succeed or fail depending on the quality of the team members. Managing and inspiring these people has evolved away from some of the harsher, old-school techniques. Instead, companies will continue to grow based on how well they treat and support their key creative talent. Those companies that show a real understanding of this shift and effectively connect with their teams will wind up developing the best products. ■

Scott Steinberg, vice president of marketing at Crystal Dynamics, has been responsible for managing and marketing GEX, the company's fast talking, top-selling, gecko lizard that has been the company's top revenue-generating product. Prior to joining Crystal Dynamics, Scott was in marketing management at Sega of America, where he was responsible for marketing and promoting Sega Genesis, Game Gear, and Sega CD software products.

PANDEMONIUM 2

Game platforms: PlayStation and PC CD-ROM

Development time (from conception thru shipping): Approx. 1 year

Size of development team (excluding QA): 22 people

Ship date: October 1997

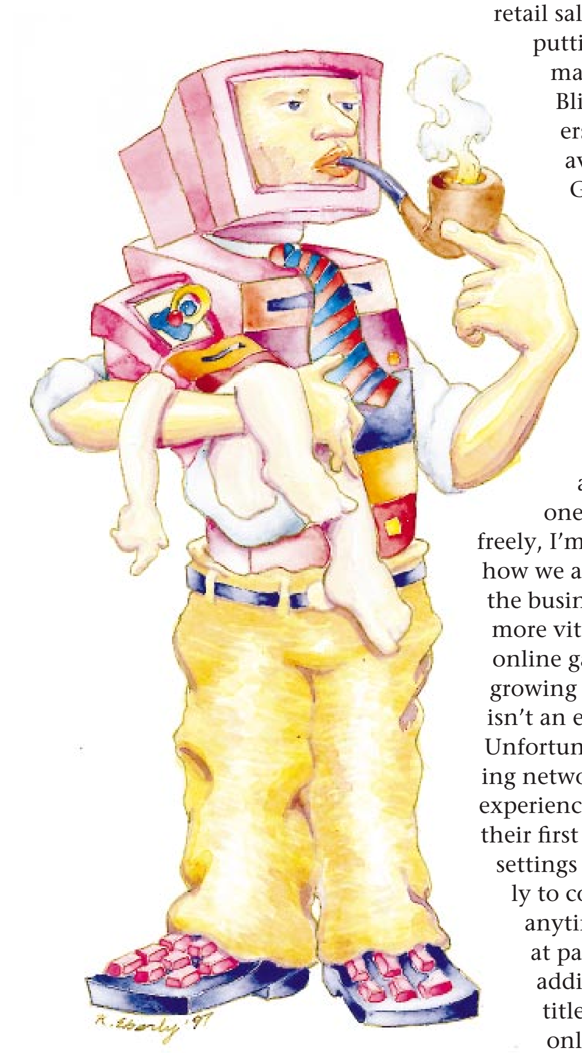
Publisher: Crystal Dynamics

Distributor: Midway

by Jack Heistand

Nature vs. Nurture: Growing up Online

The entertainment software industry is poised for great expansion thanks to the potential that online multiplayer gaming represents. The question on many people's minds is not *whether* there is a business here, but *when* this business will take off? Gamers have voted with their purchases and given clear, direct feedback to publishers of packaged goods titles: "If you don't have a network component for online play, the chances of your game being a hit plummet." The reaction among publishers right now is quite pragmatic: enable games for online play and make online play as widely available as possible to drive retail sales. Some publishers are putting up their own servers for matchmaking, such as Blizzard's Battle.net, while others are making their games available everywhere, as with GT Interactive's BLOOD.



The risk with these strategies in the long term is that free online play will become a commodity feature that all games include by default. But where's the real value coming back to developers and publishers? With everyone providing online play freely, I'm hard-pressed to understand how we as an industry are growing the business and making the market more vital. Delivering the best online game experience for the growing communities around games isn't an easy or inexpensive exercise. Unfortunately, the free online gaming networks are providing subpar experiences. Consumers obtaining their first experience in suboptimal settings such as these will be unlikely to come back to online gaming anytime soon, and will likely balk at paying for the privilege. In addition, the more broadly a title is distributed to various online gaming solutions, the harder it is for a community to put down roots.

We're all learning a lot. It's time for us as an industry to get together and determine a business model that makes sense for the consumer, the developer, the publisher, and everyone else involved. It's time for us as an industry to share learning; for distribution of that learning. The time for smoke and mirrors and the circulation of misinformation is over. In that spirit, the following is offered.

Priority One: Creating a Market

We don't have a true market yet, either in terms of the consumer audience or in terms of a broadly recognized business model. The business model must be in place before the audience really begins to arrive, or the current confusion in this market will persist. Ultimately, the two broad models up for consideration are advertising-based and the "consumer-pay" model. In the latter, any and all types of hourly, daily, weekly, monthly, or yearly "pay-to-play" formats are included. There will be many different types of these pay-to-play formats for some time, and that's okay. With regard to the ad-based business model, it's not here yet.

Ad revenue will not create any meaningful value to developers and publishers for at least three years. If there is one publisher who has gotten a royalty check from advertising of even $10,000, please stand up and show us the money. Meanwhile, many of TEN's content partners have already seen six-figure royalty payments, thanks to our subscription revenue stream.

The lowest barrier to entry for success today is to be a developer and/or publisher of online-only content. The big guys aren't focussing in on it. They can't afford to take their best talent off of predictable retail franchises and

Continued on page 63.

Continued from page 64.

send them on fishing expeditions. The best and breakthrough content will come from smaller companies like ICE Online, VR1, Multitude, and others.

Priority Two: Games and Gamers

To truly grow the online gaming market segment, developers must do a better job of addressing the needs of the audience through the games themselves.

- Communities are loyal to games first, genre second, and nothing else.
- Consumers want to compete. They want tournaments, they want national game play, and they want to understand their personal rankings relative to the other competitors.
- The best games are organic. They evolve and grow over time to reflect the needs of the community. This isn't limited to role-playing games; witness QUAKE and DUKE NUKEM 3D. Wherever possible, this organic metaphor should be extended to include and involve the user: providing a game's community with the tools to extend the game is far easier than doing it "in house" and adds a sense of involvement that sophisticated gamers appreciate.
- The technical hurdle of latency must be further overcome in order to truly grow the online gaming business. Only then will online games rise to mass market proportions with the advent of new genres such as sports simulations and fighting games. Game design should address latency where possible — either in the optimization of data transmission or by creating "cheats" and workarounds that make latency less of an issue. But equally important and often overlooked is the need to reach true consumerization of online gaming through better ease of use and installation. There is huge room for improvement in this area.

Priority Three: The Need for Standards

For most industries, to grow far beyond the "start-up" phase, some standards are a must. Microsoft and AOL are not serious about the business. The former needs to win as a publisher first; they've struggled for over five years to build only a 5% market share in gaming. Their first goal will be to try to topple EA. I'd rather be locked in a jail cell with a hungry Mike Tyson than take on EA. The business is just too unpredictable and Microsoft lacks the leverage that it typically enjoys. At the same time, AOL needs applications that drive usage for as many of their 8-10 million subscribers as possible. INN's hearts and cribbage are the focus for now. We as an industry cannot, and should not, rely on these mainstream industry leaders to dictate the evolution of the online gaming business.

The consoles are coming. The PC has had the opportunity to distinguish itself as the sole Internet access provider and make serious inroads on the console dominance in electronic entertainment. That window will soon begin to close. With the console companies, it seems highly unlikely that the publishers will have much say in the online business model. While consoles suffer from the finite lifespans of their audiences, which are not portable from one platform to another, the big winners — Nintendo and Sony — have enough critical mass if they build the communities at initial ship. What hurt the Sega Channel was its launch at the peak of the 16-bit market. By the time they had amassed 200,000 online users, there was little new content coming. Not to mention that in Sega's business model, publishers received only a few cents on the dollar. Ouch. Instituting some standards will help ensure that when the console companies enter the online gaming market in a significant way, there will already be in place established and accepted parameters and guidelines.

Database and billing administration is a nightmare. It took TEN three months after commercial launch to iron out most of the kinks. As of this writing, we've sent 16 consecutive bills out. I don't think there's an off-the-shelf application that will work here. We probably would be better served as an industry to agree on one application and embrace that.

There are wild figures being bandied about regarding the numbers of users on systems. To the extent that advertising is, or eventually can be, a significant revenue source for online gaming companies and their content partners, some official, independent auditing mechanism is necessary. While TV has the Nielsen ratings, radio has Arbitron, and magazines have the BPA, the Internet, and online gaming in particular, has no such means of validation. For the free sites out there, the numbers cited seem to reflect unique visitors who have come to the sites at least once. A better approximation of a site's "active member base" is a must if advertisers are ever to begin taking this market seriously. Currently, the best approximation of a site or service's active member base can be obtained by tracking peak simultaneous usage — the number of users a service attracts at its busiest times of day. As a rule of thumb, peak usage runs about 4% of the total active member base. This equation yields that most of the free services are overstating their active base by a factor of four or five. In total, there are about 150,000 PC games enthusiasts who have signed up to TEN, Mpath, and the Internet Gaming Zone combined; the rest are either duplicate accounts or one-time, "tire-kicking" curiosity-seekers. These estimates are reinforced by a recent IDSA study that found that 4% of the 3.5 million PC game enthusiasts have subscribed to an online gaming service.

So what does all this mean? It means that online gaming has a lot of growing up to do. Where do we begin? We must come to some agreement on the direction this market segment will take and work with some level of cooperation to move in that direction. Let's face it: Online gaming can be as significant to the entertainment software industry as the video rental business is to Hollywood. Or we can continue to muddy our own waters and watch a potentially lucrative market remain a niche play. ■

Jack Heistand is the president and CEO of Total Entertainment Network, a commercial online gaming service with three distinct revenue streams: subscriptions, advertising, and technology/brand licensing. Heistand previously served as senior vice president of corporate business development at Electronic Arts, and also held senior management positions with Hearst Magazines, where he was the founding publisher of SmartMoney *and* The Wall Street Journal Magazine.