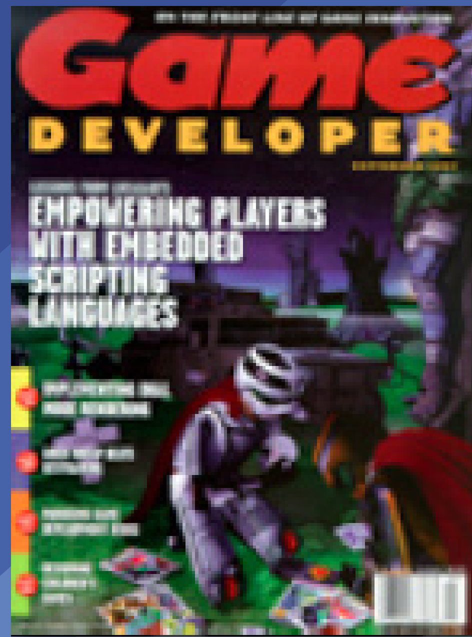
GAME DEVELOPER MAGAZINE

SEPTEMBER 1997



UBM Tech

The Wal-Mart Effect

The power that retailers have over our industry today is chilling. However, if you think we have it bad, pick up *Rolling Stone* #764 last July and read about Wal-Mart's recent antics in the music industry. The article explains how this giant chain of discount stores has risen to prominence in the retail music industry, due to the departure and downfall of dedicated music chains such as Musicland and The Wherehouse. With more than 2,000 stores, Wal-Mart now represents almost 10% of the retail music market's sales, and it's shooting to double that figure within a few years. With this much retail power, Wal-Mart has learned that it now has the ability to shape content within the music industry.

Wal-Mart is a "family values"-oriented business. For this reason, they sometime decline to carry music with objectionable lyrics or cover art. As a huge retailer, though, their decisions have convinced some musicians to conform to the chain's tastes. John Mellencamp decided to change the lyrics of one of his songs to get on their shelves. Yanni had to appeal to Wal-Mart executives to carry his Christmas CD, which showed him playing the saxophone for his naked infant in front of a Christmas tree. Musicians actually release special Wal-Mart versions of their albums, with changes made to overcome the objections of Wal-Mart executives.

I don't consider Wal-Mart's actions censorship. The company has a right to not sell any product, and the company is quick to point that out. Wal-Mart spokespeople maintain that it's up to the individual musicians to alter their albums if they want to get Wal-Mart shelf space, and that no arm twisting is taking place. It's simply their dominance as a retailer that makes artists reconsider their choice of lyrics and cover art.

My concern is whether games could be the next target of Wal-Mart's family-values crusade. As a developer, you may someday have to contend with a publisher calling the shots on game content in order to obtain an ESRB rating acceptable to the retailer. You might have to develop multiple "politically correct" versions of a game to get maximum exposure in the retail channel. Publishers, on the other hand, may find themselves taking on the role of policemen to make sure that there are no surprises in game content that could sink distribution plans with the retailer.

While the rating systems we now have are adequate for today's purposes, I question whether they are ready to bear the pressures that the above scenario would put on them. RSAC ratings are given by the developers themselves — that won't stand up to the scrutiny of Wal-Mart management. The ESRB system, on the other hand, uses three (count 'em, three) "demographically diverse" people to rate a game. (And I thought television's Nielsen rating system used a small sample set.) As the stakes go up in the ratings game, the threat of a publisher abusing our ratings systems grows larger and larger, and these two systems are not up to the task. However, more complicated rating schemes mean more administration of the system and therefore higher costs, which of course will be passed down to publishers in the form of higher rating fees.

This scenario scares the hell out of me for other reasons too. Who knows what consumers will do if a major retailer stops stocking a certain popular game? Will consumers shop elsewhere, simply shop at the same store and put up with a limited selection to choose from, or turn to catalogs or the Internet for their game purchases? In some sections of the country, Wal-Mart is *the* game retailer for A titles, so consumers may only have the last option if they want a specific title.

What are your thoughts about the current state of retail, and about the prospect of retailers dropping your title because of its content? Let me know. I'll print your stories and opinions in an upcoming issue. ■

Game Developer

ON THE FRONT LINE OF GAME INNOVATION

EDITOR IN CHIEF Alex Dunne adunne@compuserve.com
MANAGING EDITOR Tor Berg tdberg@sirius.com
EDITORIAL INTERN Alex Clark alex@mfi.com
EDITOR-AT-LARGE Chris Hecker checker@bix.com
CONTRIBUTING EDITORS Brian Hook bwh@wksoftware.com
David Sieks dave@ward1.com
Ben Sawyer bensawyer@worldnet.att.net
ART DIRECTOR Azriel Hayes ahayes@mfi.com
ADVISORY BOARD Hal Barwood, Noah Falstein, Susan Lee-Merrow, Mark Miller, Josh White
COVER IMAGE PCA Graphics

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
SALES ASSISTANT Chris Cooper (415) 908-6614 ccooper@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
CIRCULATION MANAGER Lisa Eversole
CIRCULATION ASSISTANT Glenn Wagner
NEWSSTAND MANAGER Eric Alekman
REPRINTS Stella Valdez (916) 983-6971

Miller Freeman
A United News & Media publication

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

www.gdmag.com

BIT BL

NEWS FROM THE WORLD

INDUSTRY WATCH

by Alex Dunne

DOUG LOWENSTEIN was singing the E3 blues this summer, after the Atlanta show's attendance numbers went south (no pun intended). About 37,000 people showed up this year, down considerably from last year's event in Los Angeles, which hosted about 57,000 people. What was surprising was Lowenstein's candor with the press about the decision to move the show to the deep south. In an interview with the *San Francisco Chronicle*, show organizer Lowenstein admitted that "the show belongs on the West Coast; we're acutely aware of that." Unfortunately, the group is tied to the venue for at least another year, due to a multiyear contract signed with the convention facility.

IF MICROSOFT wasn't getting the picture that a large number of game developers aren't happy with D3D and the company's less-than-stellar support for OpenGL, yet another open letter (following Chris Hecker's first letter in June) from game developers was mailed up to Redmond in July. The second letter, written by Brett Douville, pled with the company "to continue its active OpenGL development, to ship its DirectDraw bindings for OpenGL and the Windows 95 MCD driver-enabled OpenGL, and to continue to improve its implementation of the OpenGL API and its driver models by aggressively supporting common extensions and future." The letter was initially signed by 253 people, and after it went off to Microsoft and word got out, over 1,100 more people threw their support behind it. You can read the letter and see the signatories at http://www.multiweb.nl/~henk/OpenLetter/.

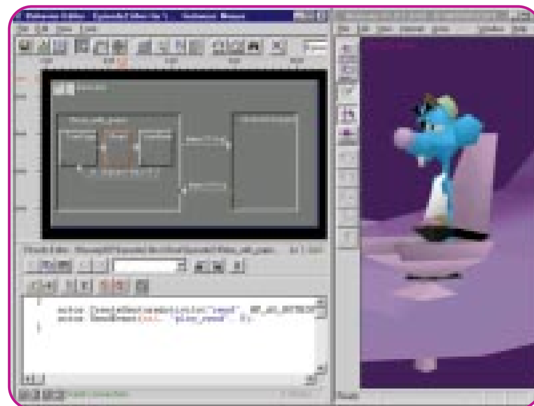
SPEAKING OF DIRECTX, the development team (which is now led by Kevin Bachus, since the departure of Alex St. John) just sent out the final version of DX5. The accompanying letter indicates that version 6 is scheduled for release next April 22 (beta available around January 1), so if you want to send the company feedback or feature suggestions, send it to directx@microsoft.com.

Creating Intelligent Characters

THE MOTION FACTORY has released fa new suite of game development tools, based on research into the fields of robotics and real-time process control. The product, known as Motivate, specializes in creating 3D characters that act intelligently. The software is based on a high-level, state-machine-like programming system that analyzes and reacts to events in a game. The Motivate development tools comprise three components:

- The Actor Editor, for importing the 3D model of a character (currently supports 3DS and MAX formats) and its associated motion capture data (Motivate currently supports the BioVision format).
- The Skill Editor, for creating the motions and gestures of a character.
- The Behavior Editor, which combines a visual programming environment with a JavaScript-like scripting language to create, modify, view, and debug game logic scripts for game characters.

The system requires that you embed a small Motivate Runtime engine in the game, and there is a Motivate Server for deploying multiplayer games over a network or the Web. An SDK allows C++ programmers to create custom player views and other extensions to the system.



The Motivate development tools, runtime engine, and optional server require Windows 95/NT. Pricing is a flat $25,000 for an unlimited use license per title, plus a $25,000 royalty for unlimited client distribution.

■ The Motion Factory
Fremont, CA
(510) 505-5150
www.motion-factory.com

Oak's 3D Chip Features Warp Speed

OAK TECHNOLOGY introduced its WARP 5 (Windows Accelerator and Rendering Processor) 3D chip, yet another entry into the growing market of 3D acceleration hardware. Oak's WARP 5 chip combines 2D acceleration, video, a dual-clock generator, and RAMDAC functions on a single chip. Oak uses a region-based rendering approach, promising better performance for highly dynamic game scenery such as that found in flight simulators. The key to the way the WARP 5 renders is in the way it partitions the screen into subregions, and then processes and renders them one at a time. This allows the image's Z values and antialiasing information to be stored on the chip at the sub-

Scene rendered with WARP 5.

ASTS

OF GAME DEVELOPMENT

pixel level — no external storage is required. Oak claims that WARP 5 can process more than 50 million pixels per second, with perspective correction, trilinear MIP-mapped textures, antialiasing, Z-buffering, Gouraud shading, translucency, and fogging. The WARP 5 architecture supports Microsoft's Direct3D as well as Argonaut's BRender 3D APIs. The WARP 5 will begin sampling to OEMs in July, and volume production will begin in the fourth calendar quarter of 1997. It will be available for $35 in OEM quantities.

■ Oak Technology
Sunnyvale, CA
(408) 737-0888
www.oaktech.com

A Low Bandwidth Voice, Audio Solution

VOXWARE announced the the Voxware Compression Toolkit (VCT) 1.5, a speech codec technology that is especially good at transmitting voice audio over limited bandwidths — think Internet games, particularly if you want to build in some type of simultaneous chat. New to version 1.5 are stereo streaming technology and VoiceFont voice-altering technology. Voxware's VCT provides improved sound quality over limited bandwidth channels. The VCT's MetaVoice speech codecs don't compress audio; they transmit instructions for recreating the voice on the destination platform. The results are accurate and require relatively small bandwidth and storage space. If you're into helium-inhaling effects, the product also incorporates VoiceFont 2.1 technology, which allows you to alter the sound of the human voice. People's voices can be sped up or slowed down, made to sound like several people, and otherwise manipulated in real time to create special effects. The VCT's MetaSound codecs let you deliver music and audio effects while requiring small storage and bit-rates. Developers can license any combination of the MetaVoice and MetaSound codecs, as well as optional VoiceFont and other API technologies. VoiceFont technology is only available to licensees of MetaVoice. Mono and stereo MetaSound codecs are priced separately.

■ Voxware
Princeton, NJ
(609) 514-4100
www.voxware.com

Triple Threat Tool for PowerAnimator

ALIAS|WAVEFRONT has announced a new set of PowerAnimator add-on tools specifically aimed at game developers. The product, called PowerAnimator Worlds, helps developers create high-performance 3D scenery. The performance gain comes from three tools in the product. First, a BSP Tree Builder lets you create, preview, and display BSP trees and edit them for your game environment. Second, the Terrain Builder lets you import images or geographical data, convert it into polygonal wire files, and edit the resulting model's color and details. Finally, there is a Level of Detail (LOD) Manager for managing the amount of detail displayed in polygonal models as they get closer or farther away from the viewer. PowerAnimator Worlds is available for $4,995 and runs on the SGI platform.

■ Alias|Wavefront
Toronto, Ontario, Canada
416-362-9181
www.aw.sgi.com

IF YOU ARE TARGETING the perennially lucrative market of lazy federal employees for your next game, you'd best make yourself aware of the latest push on Capitol Hill to wipe games off of computers in federal agencies. Republican Senator Lauch Faircloth introduced a measure to do exactly this, after finding out how easy it is to switch from games to work on a computer. In his statement, the Senator remarked, "The taxpayers don't need to be paying the salaries of people who are playing games while on official time." No word on how the Senator's comment affects Kenneth Starr's Whitewater hearings….
HARD TIMES are taking their toll on TEN, which recently trimmed its workforce down from 95 to 80. With more and more competition in the online games market, the company better have some good cards up its sleeve. Adding to TEN's headaches, Activision just announced that it's going to launch an online gaming service as well, which will host all of the Activision's multiplayer games. Their service should be ready by this fall, and will include player matching, player rankings, automatic game updates, and chat rooms. Hmmm. Sounds like a familiar service.
UNCLE! Yes, Matsushita has surrendered — at least for now. The company announced that it won't try to challenge Nintendo in the 64-bit arena. Yoichi Morishita, Matsushita's president, admitted that its rival Nintendo grabbed too large of a lead in the market, and that the company was scrapping its plans to introduce a 64-bit console by the end of the year. Looks like Sony and Nintendo have the run of the house for the foreseeable future. Also in console news, Sega broke news to 3Dfx that the next generation Sega console system would not use a 3Dfx graphics chipset. 3Dfx had been working under contract with Sega since last March to develop and license a proprietary chipset for the new Sega console. Sega had funded development of the chipset as well. Looks like the lawyers could get involved, too, as 3Dfx president and CEO Greg Ballard pointed out in a not-too-veiled threat: "We are disappointed with this notification, and believe that it is without legal justification…."

All I Want for Christmas '98 Is a Hardware Accelerator That Doesn't Suck

At the 1997 Computer Game Developer's Conference, I hosted a roundtable called "3D Acceleration, One Year After" to discuss the experiences that hardware and software that the developers had with 3D accelerators over the previous year. At one of my sessions, a key issue was that "hardware vendors don't know what software developers want in an accelerator." At the urging of the crowd, I started writing a list of things that game developers absolutely need in order to develop cutting-edge titles. Unfortunately, I never finished the list because we got distracted by the Direct3D versus OpenGL monster.

So, this month I'm going to try to outline what I think are reasonable requirements for baseline hardware acceleration — the minimum reasonable hardware accelerator for a game shipping for Christmas of 1998. And unlike some product manager weasel at a board company, I'll actually explain why some of these features and targets are important.

In this space next month, I'll discuss my wish list for hardware accelerators that are aiming for a 1999 or later release date (that is to say, those accelerators beginning development when this article is published). In all likelihood, accelerators coming out in that timeframe should be reaching the knee of the curve for the triangle rasterization model of doing things.

A Baseline

Because of space constraints, I can't explain what each feature does. If you don't know what Z-buffering or bilinear filtering is, I'd recommend browsing the web for a good tutorial on 3D acceleration.

The first thing I want to establish is a baseline set of features that any self-respecting 3D accelerator should possess, no matter how crappy the hardware is. Any board worth its silicon should have a feature set that includes:

- Gouraud shading
- Dithering
- Subpixel/subtexel accuracy
- Perspective-correct, bilinear-filtered texture mapping
- RGB rendering.

Unlike a "product manager weasel" at a board company, Hook explains what features 3D hardware has to support.

Barely Beyond The Basics

Okay, now for those things that are "slightly beyond the basics." Many of these features are implemented poorly on a lot of today's 3D accelerators, but should be fairly robust by Christmas 1998.

To all you hardware folks reading this, please remember that giving developers more options is a good idea. While this guideline may seem absurdly obvious, it seems to have escaped many of the engineers who design the chips that we're stuck with supporting. Just because we can't come up with a use for a particular feature today, doesn't mean that we won't tomorrow.

Z-BUFFERING. Can you believe some hardware companies actually thought no one would use Z-buffering? At the risk of stating the obvious, Z-buffering is necessary! We need more bits of depth resolution — 16-bits of Z is barely enough for today (GLQuake suffers from Z-aliasing artifacts even with 16-bit Z-buffering), but it's a reasonable baseline.

Some minor points to consider: The ability to selectively enable writes to the Z-buffer while still doing depth comparisons is vital for things such as rendering translucent surfaces. All of the Z-comparison tests are absolutely necessary.

ALPHA BLENDING. If you read last month's

column, ("Multipass Rendering and the Magic of Alpha Blending," August 1997) you probably realize that I think alpha blending is a vital feature for tomorrow's games. A look at many Nintendo 64 titles will show you why — the translucent water effects in SUPER MARIO 64 or WAVERACE 64 just wouldn't work without alpha blending. Several titles for the PC also use alpha blending to great advantage — TOMB RAIDER (Core/Eidos) and GLQUAKE both use alpha blending to render water.

A basic hardware accelerator should support all blending factors. We need to get away from the notion that alpha blending is only used for translucent surfaces. Different alpha blending factors can be used for achieving other kinds of effects — GLQUAKE uses alpha blending to render two-pass lighting maps.

ALPHA TESTING. Alpha testing is vital for doing sprites and partially transparent texture maps. Alpha blending alone doesn't suffice, since we want a feature that will actually reject a pixel instead of just rendering it transparently (there's a very important distinction between the two — the former won't write to the Z-buffer if a texel is transparent, but the latter will, causing all kinds of weird bugs). As with Z-buffering, all the alpha test functions are vital, so no skimping.

PERFORMANCE. Obviously, game developers want as much performance as possible, so assume no upper bounds; but, it would be nice to be able to expect a certain minimum level of performance. For Christmas 1998, it's not unreasonable to ask that a decent accelerator have at least 50 megapixels/second of usable performance. Thankfully, slower fill rates can be made up for by switching to a lower screen resolution.

To make life easier for developers, it would also be nice if the inverse relationship between enabled features and performance wasn't so... obvious. There's nothing more frustrating than getting a new accelerator that claims a bazillion megapixels, but by the time you enable all the features, the accelerator is outclassed by a drunk squirrel with an Etch-a-Sketch.

There's nothing more frustrating than getting a new accelerator that claims a bazillion megapixels, but by the time you enable all the features, the accelerator is outclassed by a drunk squirrel with an Etch-a-Sketch.

From a more technical side, it would be nice if hardware accepted both `D3DTLVERTEX` data and OpenGL-style vertex information (raw floating point) directly. On-chip triangle setup is also a necessity so that the CPU isn't loaded down — but make sure your triangle setup is fast enough so that it won't be a liability when newer CPUs are released.

LINE AND POINT RENDERING. Line and point rendering have largely been neglected by the hardware acceleration community. To be honest, I haven't missed it. (Keep in mind, just because I don't think line and point rendering is useful doesn't mean it isn't!) However, when developing tools such as level editors and modeling programs, the ability to render antialiased lines and points becomes vital. Hardware designers have to keep in mind that 3D acceleration is slowly going to be adopted by applications as well as by games.

FOG. Fog is nice. Fog is even better if you can use it without worrying about some cheesy mutual exclusion by, or dependency upon, another technique (such as alpha blending). I'll talk about this later when I address flexibility. Because of my work at 3Dfx, I have a bias towards fog tables — the fog curves that Direct3D and OpenGL use simply aren't nearly as useful as a generalized fog table.

Another concept I find intriguing is vertex fog, that is a "fog iterator." This can be approached in one of two ways : an iterated fog blending coefficient that is used to blend between the pixel color and a global fog color, or an iterated fog color (RGBA) that is used to blend between the pixel color and the iterated fog color. The iterated fog color approach is far more generally useful, since it allows an arbitrary blend between the computed color and a separate iterated color. It also requires significantly more gates to implement and thus is more expensive.

APPLICATION-MODIFIABLE GAMMA TABLES. When game developers joined the great migration from VGA Palette Land to Hardware Accelerator RGB Land, they lost the cool palette tricks that VGA permitted. A lot of these palette tricks simply enabled flare/flash/saturation/wash-out effects. As a stop-gap replacement, developers can use full-screen alpha blended polygons to achieve similar effects. However this technique isn't the greatest thing for performance. An alternative to full-screen alpha polygons is a programmable gamma table — the Win32 API even exposes this through the `Get/SetDeviceGammaRamp` functions. Unfortunately, very few drivers implement these functions.

Hardware companies should implement those Win32 API calls in their drivers. With the Rendition Verite version of QUAKE (VQUAKE), we used the Verite's application-accessible fog tables to achieve flash and saturate effects. So if you build it, we will come. I promise.

FLEXIBILITY. Accelerators need to be flexible — I want to be able to mess around with all the different features that OpenGL and Direct3D expose, but at the same time I don't want the accelerator or driver to explode because it can't do feature X at the same time as feature Y. Reasonable independence between features is absolutely vital if game developers are to have the freedom to follow their imaginations.

So no weird mutual exclusions or dependencies, please. Game developers hate hearing things like, "You can't do perspective correction if texture wrapping is enabled," or "You can't fog unless you're texturing," or "You can't

alpha blend and fog at the same time." Unreasonable mutual exclusions or dependencies are evil.

Here's a simple test: if you're a developer relations engineer, and you find yourself telling developers anything resembling the following two phrases, you're screwing up:

We need hardware accelerators that give us at least 4MB of texture RAM, preferably more. Even 2MB is barely sufficient, assuming texture download performance isn't ghastly.

"You can't do [feature A] unless [some other feature that has nothing to do with feature A] is enabled."

"You can't do [feature A] if [some other feature that has nothing to do with feature A] is enabled."

PALETTED TEXTURES. In general, I'm against paletted anything, but paletted textures do have their uses. Palettes provide a form of cheap and easy compression, and even today a lot of hardware accelerators support four-and eight-bit paletted textures.

NO TEXTURE DIMENSIONALITY LIMITATIONS. Hardware accelerators need to accept nonsquare texture maps (such as, 128×64) with arbitrarily high aspect ratios. Also, there should be no arbitrary bounds on maximum texture size — if there is enough texture RAM to support a 2048×2048 texture, then allocating a texture of that size should be possible.

Obviously, nonsquare textures are useful when trying to texture map nonsquare images (duh); however, some hardware designers don't support nonsquare textures. They think developers should just pack several rectangular textures into a single nonsquare texture and have texture coordinates adjusted accordingly. This actually works, except when you need to use repeating textures. For this reason, we need rectangular textures.

Unlimited texture map sizes are handy when an application wants to manage textures very precisely. For example, a game could allocate a 1024×1024 texture and put a bunch of smaller textures within it, assuming that texture wrapping was not necessary. This reduces state changes, since only a single texture is being used at all times, and also reduces texture RAM fragmentation within the driver, since only a single large block of RAM is being allocated.

SUPPORT ALL TEXTURE FORMATS. A good 3D accelerator should support every single fundamental texture format that OpenGL and Direct3D support, within the capabilities of its texture storage (not supporting 32-bit texture formats is acceptable if a hardware accelerator only stores 16-bit textures). That's a hard and fast rule — developers don't want to deal with having to convert art on the fly, nor do they want to rely on something like textures with alpha channels only to discover that one particular accelerator doesn't support textures with alpha.

STRAIGHTFORWARD TEXTURE ALLOCATION SCHEMES. Texture memory should be allocated in a straightforward, uncomplicated manner. There shouldn't be any unreasonably large minimum texture size — I've heard of some hardware accelerators that specify a minimum texture size granularity, which means small textures are extremely inefficient. For example, an accelerator might have 1MB of texture RAM that's divided into 250 discrete 64×64 chunks — even if you use a 16×16 texture, it will consume one "slot" that is 64×64 texels in size, wasting over 90% of the allocated texture RAM.

TEXTURE CLAMPING AND WRAPPING. Hardware accelerators must allow each texture coordinate axis to be clamped or wrapped. There should be no constraints limiting the availability of clamping or wrapping (for example, some accelerators won't allow perspective correction if texture wrapping is enabled).

MORE TEXTURE RAM. Having limited texture RAM sucks. Bad. We need hardware accelerators that give us at least 4MB of texture RAM, preferably more. Even 2MB is barely sufficient, assuming texture download performance isn't ghastly. I don't know if AGP is the answer here — until I see a hardware accelerator that demand loads textures across AGP, I'll reserve judgment. Somehow, we need to find a way to get at least 4MB of texture RAM to the developer, if not more.

If a 3D accelerator isn't going to provide a lot of texture RAM, then at the very least it will need to provide very fast texture download performance, preferably in an asynchronous fashion using bus mastering.

Texture compression is one way of solving this problem, but if it takes up any CPU cycles, it's not going to be worth it. Games can't suffer inconsistent performance on one accelerator simply because every texture download or allocation kicks off a slow texture compression operation on that accelerator. You could potentially precompress a texture during texture allocation time, but this presumes that you're not dynamically generating your textures. Since procedurally generated textures are going to become more common in the future, I don't see texture compression really helping us out.

High Hopes

To be honest, I think all of the above features will be implemented in most accelerators by the early months of 1998, but there will probably be some stragglers in the market who won't get their acts together until Christmas 1998. If we can guarantee at least the above features in all accelerators by Christmas in a year and a half, game developers should be smiling. ■

Brian Hook was formerly an engineer with 3Dfx Interactive and a contractor in the games and semiconductor industries. Today, he is a proogrammer for a small game company known as id Software. He'll be working on QUAKE 2 and TRINITY. You can contact him via e-mail at bwh@wksoftware.com.

ADDING LANGUAGES TO GAME ENGINES

OK, I'll admit it. I'm a lazy programmer. Whenever I can get away with it, I love to get other people to do my job for me. And as a programmer focusing on gameplay and simulation, life keeps getting more and more complex. Just as the things we used to toil away at for hours – sound mixing, 3D rendering, interrupt processing – are getting easier and easier, simulation programming is getting harder and harder. Now that we have DirectSound and DirectDraw, when can we expect DirectGameplay?

I can remember a time not long ago when bits were the solution for everything. Back then, when a designer

BY ROBERT HUEBNER

wanted some cool new feature, the solution was to write some code, find the next free bit in the `CoolEffectsFlags` bit vector, and recompile. Once the designer enabled the bit, the new feature emerged, ready for action. But lately, I've run out of bits.

The problem is, users are demanding more interactivity and unpredictability from their games. They aren't satisfied with 10 types of weapons when your competitor has 20. Moving platforms aren't sufficient if some other game has rotating platforms. So what's a lazy programmer to do?

Scripting languages have been an integral part of games for many years. Long before action games ran out of bits, adventure game authors recognized the need for scripting to cope with the massive number of possible interactions in their worlds. SCUMM (Story Creation Utility for Maniac Mansion), one of the original adventure game languages, has survived virtually intact until the present day, and is still used for games such as MONKEY ISLAND 3. As other game genres such as action, simulation, and strategy become more complex, they too are incorporating scripting systems.

The best way to stay competitive in the race for bigger and better games and game engines is to keep the engine as flexible, expandable, and robust as possible. An internal scripting language allows you to create a separate, crash-proof environment inside your game engine. This protected virtual machine executes the complex and frequently changing gameplay code, protected from the "real" machine running the game engine. By partitioning the code in this way, you significantly reduce the complexity of the core engine, resulting in fewer bugs and a more robust game. And since a language system is far more flexible than a collection of "canned" effects, your engine will be able to do more interesting things, even things you didn't originally anticipate.

Using a script language allows the engine programmers to focus on what is important to them — refining and optimizing the core technology of the game — while the game designers can handle the gameplay details. If the language is simple and well-designed, nonprogrammers can implement their designs directly in the script language without endangering the core engine code or involving the engine programmers. And since programmer time on a project is usually limited, recruiting designers as scriptwriters allows more of the original design to be realized, resulting in a more interesting final game. In fact, most designers jump at the opportunity to directly implement their ideas in script, even when it requires learning a new language.

The Snowball Effect

Over three years ago, the original team developing DARK FORCES (the sequel, with which this article is concerned, is shown in Figure 1) took the unconventional step of implementing some of the important game systems using a special parsed opcode language called INF. INF (which, as far as anyone on the original team can recall, doesn't stand for anything) was used for simple tasks such as moving sectors and elevators around, tracking the mission goals, and generating new enemies. INF didn't require any complex parsing because the format was simple and direct — the script equivalent of assembly language. One of the design goals in creating the sequel was to expand and enhance the INF language, making it more powerful and user-friendly.

One of the main complaints from level designers on the original project was that INF required the use of a lot of specific flags and opcodes to enable various features. A common fixture near the designer's workstation was a stack of pages affectionately known as the "Zen and the art of INF." The first draft of a replacement INF retained its basic structure, but translated the numerical codes and flags into text so this "bible" would no longer be necessary.

One day, someone suggested that if the language was expanded slightly, it could take on the added responsibility of scripting the results of powerups, which were handled in-engine in the original game. Shortly after making these extensions, someone else suggested that it would be nice to add some simple math and conditional opcodes to the language, and these were also added. And so it went, for a period of weeks, as more and more systems were absorbed into the rapidly expanding snowball that was INF 2. It became

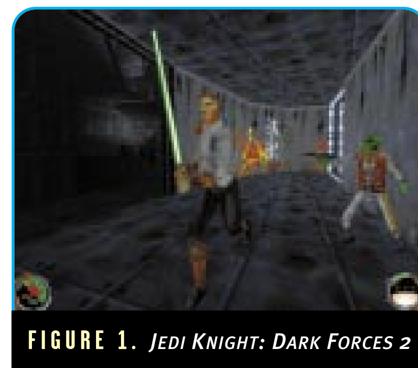


FIGURE 1. *JEDI KNIGHT: DARK FORCES 2*

clear that there was a need for a more flexible, all-purpose scripting language, and the snowball transformed into COG (which, true to the spirit of INF, also stands for nothing).

The Paths Not Taken

There were two primary goals for our language. First, the syntax should be powerful enough to offer complex loops, conditionals, and nesting, but familiar enough to be learned and used by a nonprogrammer. Second, the language must execute quickly enough to be useful in a real-time action game.

The first stop on any language shopping trip should be the catalog of free compilers at http://www.idiom.com/free-compilers/. Here, you can find dozens, if not hundreds, of existing scripting libraries that can be linked with your application. Each of these has various advantages and disadvantages. Some are very simple, such as LISP or FORTH, while others are quite complex, such as JAVA, Tcl, or LUA. Most of these languages are also completely free, the products of university or government research projects. The main disadvantage of using a ready-made language is performance. Many of the languages are at least partially interpreted, and many do not provide source code for the speed-critical execution kernel. If development time is the primary concern, or if your application is less dependent on fast execution, there are several excellent possibilities here.

Since execution speed was a primary concern, the possibility of expanding the game engine via dynamic link libraries (DLLs) instead of a script language was considered. The advantage in execution speed was clear, but using DLLs would have made it difficult for

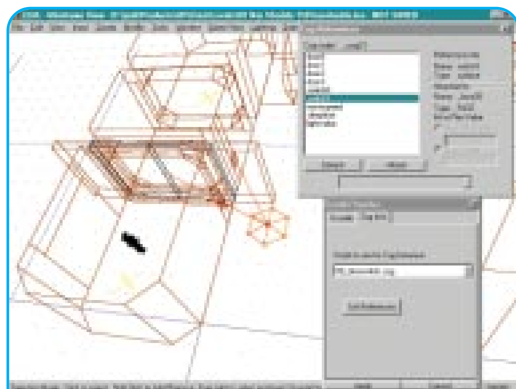


FIGURE 2. *Jedi Knight's level editor LEIA.*

the game designers to use the language directly. Even though we felt comfortable introducing them to a limited C syntax and structure, we didn't want to take the further step of introducing them to the complexities of compilers, build environments, linking, and so on.

The final option, and the one that we eventually implemented, was to create a custom language execution kernel and parser. The speed issue was addressed by performing the important, time-critical operations in native code and exporting these support functions to the language system as COG library functions. These library functions could be augmented via DLLs, which gave the advantage of native-code speed with the ease-of-use of a custom language.

COG

The rest of this article focuses on the language problems and solutions that we used in creating the 3D action-adventure game JEDI KNIGHT: DARK FORCES 2 for the PC.

For the JEDI KNIGHT language, christened COG by the designers, we chose to implement a custom, compiled language that closely resembled the syntax of C. Using the C syntax as a starting point, we removed most of the obscure keywords and constructs and even removed some fairly major portions of the language dealing with function declarations and switch statements because they were significantly more complex to parse and execute than the rest of the language. We chose the C language as a starting point because of its familiarity and the wealth of books and tutorials available teach the language to nonprogrammers.

Just as in C, the syntax of the COG language is less important than library of functions at its disposal. The COG library provides about a hundred different functions to the author, ranging from environment manipulation commands to information queries. The author uses these functions to control the game environment while using the language syntax to provide branching and looping control.

The game engine executes the scripts in an event-driven manner. For example, when two objects collide with each other in the physics engine, any COG scripts linked to either object receive a "touched" event. This event contains parameters that allow the script to identify which objects were involved in the event and the type of event that occurred. Based on this information, the script can manipulate the game state in whatever manner it wishes, or can simply ignore the event. COG scripts can also contain links to each other, which enable them to exchange messages. These events make up the primary interface between the engine and the language system.

There are additional messages that are delivered directly to the COG script rather than through the objects to which a COG script is linked. A `startup` message is sent to each COG script at the start of a level, and a `respawn` message is sent each time the local player dies. Each game object also has the ability to set a repeating `pulse` event or a one-time `timer` event to be delivered at some point in the future. This allows a combination of event-driven and scheduled execution.

Because we removed the standard C syntax for function declarations from our language for simplicity, each script is organized much like a large switch statement. The entry points into the code for various types of events are labeled using the standard C label syntax. Also, because COG expanded on the standard C variable types with the addition of game-specific resource variables (sector, thing, sound, and so on), the script variables are declared in a special header. The level editor (LEIA, shown in Figure 2) also reads this header so it can display the symbols to the designers and allow them to view edit the symbol values.

Execution Model

Each script that exists in a level is linked to any number of other objects in that level: walls, enemies, doors, other COG scripts, and so on. COG scripts execute as separate virtual machines, each with its own variables, stack, and execution pointer. Because of this, COG scripts are protected from each other. One badly written COG script can only affect itself and the objects to which it is linked. Each script is a separate resource that is loaded along with a game level. A single script can be placed in a level multiple times, with each placement having its own isolated environment.

A sample COG script is shown in Listing 1. This script creates an animating neon sign which, if it is damaged, will explode in a shower of sparks. Symbols not marked local can be modified directly in level editor tool. The

TABLE 1. *Sample COG event messages.*

Message	Description
Touched	An object or surface was touched by another object. References to both collision participants can be retrieved.
Entered	For sectors, called each time a new object enters the sector
Damaged	Called whenever the object would take damage from weapons or explosions. References to the cause of the damage and the type of damage are provided to the handler.
Created	Called on a new object when it if first created
Killed	Called when the object is about to be removed from the game
Crossed	Called for an adjoin plane whenever an object crosses it
Arrived	Called when a moving object reaches its destination
Timer	A timer event set by the script has expired
Sighted	An object is seen by the player for the first time

LISTING 1. *Sample COG script.*

```
# 00_neonsign.cog
#
# this cog will cycle through frames 0-(lastFrame-1), at framerate fps
# if damaged, it will go to frame lastFrame and stop, create sparks and sound

symbols
        message     startup
        message     damaged

        surface     sign                                mask=0x448
        float                       fps=2.0             desc=speed of anim
        template    sparks=+sparks  desc=created when shot
        sound                       exp_sound           desc=played when shot
end

code
startup:
        // Start the animation looping but skipping the first 2 frames
        SurfaceAnim(sign, fps, 0x5);
        return;

damaged:
        if (GetWallCel(sign) == 0)
                return;

        StopSurfaceAnim(sign);

        if (exp_sound)
                PlaySoundPos(exp_sound, SurfaceCenter(sign), 1.0, -1, -1, 0);

        SetWallCel(sign, 0);
        CreateThing(sparks, GetSourceRef());
        return;
end
```

`desc=` field tells the editor what descriptive string to display when the designer is editing that variable.

Access Control

One important decision made with COG was to disallow direct access to internal engine variables and structures from the scripts. If a COG script wishes to examine or modify these internal variables, it can do so only via library function calls. This is an important step in making the language crash-resistant. If a COG script could directly manipulate variables in the engine, there would be nothing to prevent badly written or out-of-date scripts from wreaking havoc with other systems. By requiring the use of access functions, any amount of validity checking and network synchronization can be added without affecting the scripts themselves. This requires a little extra work for the language programmer, since more functions will have to be written, but it pays off in terms of code stability down the road.

The COG library functions are actually just C function pointers that are visible to the COG scripts as global symbols. When the execution kernel encounters a call to one of these functions, it jumps to the native C code. The C code then calls language support functions to retrieve its arguments from the stack and return the results of the call back to the language. Since the functions are in native code, they execute significantly faster than the script language itself. For this reason, frequently performed tasks are written in C and called as library functions. Table 2 gives examples of the types of functions contained in the COG function library.

Compilation

For the script code to be executed as efficiently as possible, it must be translated from the text source code to some internal representation that can be executed quickly. This process is called compilation, and the compilation of our language source is just a simplified version of what a normal compiler does to translate source code into native machine code. Instead of producing Intel or PowerPC opcodes, we produce our own virtual machine opcodes.

The language's virtual machine is a type of simulated CPU. For COG, we use a very simple model called a "stack machine." The stack machine gets its name from the fact that it performs all operations on a single stack. Anyone who has used an HP calculator will be familiar with the system. To add 5 and 10 on a stack machine, we would execute the opcodes

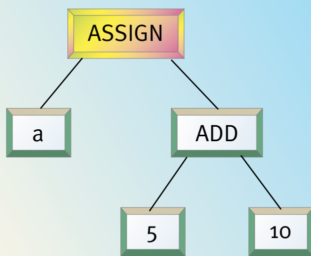
```
Push 5
Push 10
Add
```

The stack machine contains very few opcodes, making it simple to implement and efficient when executing. Our goal is to quickly compile the source code written by the designer into our custom stack machine opcodes. Any valid sequence of commands in the COG language can be broken down into these

TABLE 2. *Sample COG library functions.*

Function	Description
StartAnim	Starts a page-flipping animation on a surface, sprite, or material
SectorThrust	Sets a thrust force for a sector
SetThingFlags	Sets bits in the thing's flag field
GetCurSector	Retrieves a reference to the sector a thing is currently contained in
CreateThing	Creates a new thing in the world
PlaySoundThing	Plays a sound spatially linked to the position of a thing
SetTimer	Sets a timer event for some future time
PlaySong	Plays a redbook music track
AISetTarget	Sets the target an AI object is attacking
AISetMode	Sets the mode of an AI object
MoveToFrame	Moves an object along a path to a specified position, used for moving doors, elevators, and so on

FIGURE 3. *Simple parse tree.*



basic operations, just as normal C code can be translated into the basic opcodes of your target CPU.

The COG compilation process happens in two steps. First, the code is broken down into its relevant language parts or tokens. COG tokens, just like C tokens, include all the language keywords (**if**, **then**, **else**) and operators (**+**, *****, &&). This stage of compilation is called lexical analysis or "lexing."

The second part of the compilation process involves taking the tokens from the "lexer" and assembling them into the syntax of the language. This is a more complicated process and is based on a formal specification of the language. The formal language specification defines in detail every possible expression that can be constructed with the language in a recursive format. It seems a little awkward at first, but becomes clear after some study. For example, the formal definition of an addition operation is

(additionExpression) : (expression) + (expression).

This defines an addition expression as two separate expressions separated by the "+" token. Since the addition expression is just one of the many possible definitions of the more general "expression," you can see how the processing the language quickly becomes a recursive problem. The lowest level of the specification — the "atoms" of the language, so to speak — are the constants and variables.

Since parsing the language is a recursive problem, we build a tree to represent the structure of the source code as it is being parsed. As each language construct is recognized, we add it to the tree. The type of expression we recognize determines the structure of that small part of the parse tree. When the tree for the entire function or source file is completed, we can simply traverse the tree in depth-first order and create the stack machine opcodes that we will later execute.

Returning to the simple addition example, our completed parser should construct the parse tree in Figure 3 for the source code **a=5+10;**

Because the language parsing is done recursively, the parse automatically handles normally tricky problems such as nesting and order of operations automatically. When the code **a=(5*2) + (a^2)** is parsed, the parser will recognize the subexpressions **5*2** and **a^2** first, and will pass the completed parse trees for these subexpressions to the code that creates the tree for the addition expression, resulting in a single tree for the entire expression.

The most complex expressions to parse are those involving loops and branches. These expressions require the generation of code using the branching opcodes, which means the parser must know the address to which it needs to jump. For example, to generate code for **if <condition> then <expression>**, the parser must know the address of the code address immediately following the **expression** subtree in order to generate a **GOFALSE** opcode to jump to this code if the conditional fails.

The trick to generating code for these branches is to generate code in two passes rather than one. The first pass, known as "backpatching," doesn't generate code, but simply counts the number of opcodes produced by each node of the parse tree. During this first pass, as each node is encountered while traversing the tree, the code address (index into the array of opcodes) is noted both before the opcodes from the node are added and after. After this first pass, each node now contains the code address just prior to and just following its own subtree's code. Now, on the second pass, the branches that were previously expressed in terms of **end of node <expression>** can be expressed as actual code addresses.

This is obviously a complex topic that we have examined only superficially. For more information on the theory behind parsing and the use of parse trees, the standard text is *Compilers: Principles, Techniques, and Tools* by Aho, Sethi, and Ullman.

The Silver Lining

Fortunately, it's not nearly as difficult as it sounds. Writing compilers is an old and well-established science. There are numerous of tools to simplify the creation of an efficient parser. In fact, this is one of those rare computer science problems that can safely be called "solved." The parsers generated by the compiler tools are consistently more efficient than what a human programmer could create because they deal with the parsing problem as a complex state machine. Even a simple language specification results in a state machine with so many possible states and transitions that a mere mortal programmer would be driven (or bored) to tears.

One free tool called "lex" is commonly used to generate C code implementing a lexical analyzer based on a user-supplied grammar specification. Since COG follows the C syntax, we modified an existing free C lex specification file from the Internet to create the lexer for the language.This ANSI-C lex specification can be found at http://www.cis.ufl.edu/~fryman/c.lex.spec.html.

TABLE 3. *Sample COG "stack machine opcodes.*

Opcode	Explanation
Push	Pushes a constant of symbol onto the execution stack
Pop	Pops the next value off the execution stack
GoFalse	Pops the top stack value and jumps to a new execution address if it is equal to 0
Go	Jumps to a new execution address
Stop	Stops execution
CallFunc	Pops the next value from the stack as a C function pointer and calls that function
Add	Pops the next two values from the stack, adds them, and pushes the result onto the stack
Assign	Pops the next two values off the stack, and assigns the value of the second value to the variable contained in the first

Similarly, another free tool called yacc (for Yet Another Compiler Compiler) can be used to transform a formal language specification into a C module. yacc and lex are designed to work together, so the resulting source code modules can simply be compiled and linked to create a fully functional parser. For COG, the same Internet site yielded a full C grammar specification for yacc, which was trimmed down to our needs and used to create the parser module. The URL for the ANSI-C yacc framework is http://www.cis.ufl.edu/~fryman/c.yacc.spec.html.

The resulting compiler has all the tools needed to break down the source code and recognize the language syntax, but it is still your responsibility to write the "hooks" that tell the compiler what actions to perform when the language is recognized. These hooks are what enable us to build our parse tree. By inserting this parse-tree building code, along with some code to manage the allocation and definition of the language variables, we were able to create a C-subset compiler in about a day.

Both lex and yacc are available in many forms and permutations; some free, some not. And while lex and yacc are the most common compiler tools, there are several others including full-featured language construction environments such as VisualParse++ by Sandstone Technologies. Whichever tool you use, the end product is the same — a stream of opcodes that can be executed quickly and efficiently by your virtual machine. Check out the free compiler catalog mentioned earlier for links to these and other useful language tools.

For more specific information on the lex and yacc tools, check out *Lex & Yacc* by Levine, Mason, and Brown in the O'Reilly & Associates UNIX Programming Series.

Putting It All Together

The finished parser, developed using the free tools mentioned above, is incredibly fast and flexible. In a typical JEDI KNIGHT level, there are about 50 different script files that need to be parsed and compiled into our opcode format. On a typical machine, all these scripts compile in well under one second. For this reason, we decided against using an external compiler and instead load the source directly when loading the level. This improves the turnaround time for testing script changes, since the designer can quickly edit the script code and reload the level to test changes.

One aspect of the scripting system that proved critical for our project was the integration of the scripts and the level editor. The level editor not only allows the designer to place scripts into a level, but also ensures that the various links in the script are correctly assigned and remain correctly linked as the level changes. When a designer places a script resource into a level, the editor scans the header of the script to determine what variables can be assigned externally and places a graphic representation of the script resource in the level. This icon has a spatial location in the level, although its location isn't important to the script. Typically, designers place scripts near the objects to which they link. Once the script is placed, the designer can bring up the property dialog for that script and view and change its assignments. If the script contains links to other things or surfaces, these are assigned by clicking on the correct type of item in the level and clicking a link button. Links between COG scripts and world entities are shown graphically in the editor by connecting them with lines. If the script contains resources such as sounds or bitmaps, a pull-down menu displays the possible choices.

Another area that should be addressed is debugging. There are two main issues here — debugging syntax parsing problems, which normally is solved by adding better error-reporting and recovery code into the parser, and run-time debugging, which can take many forms. One possible method of run-time debugging of scripts is to allow the user to trace execution. In JEDI KNIGHT, the designer can enter a console command to turn tracing on for a specific COG script, which will cause that script to output debugging information each time it executes. A more complete system would allow for single stepping through script opcodes. The real challenge, which has not yet been addressed in COG, is to allow for "source-level" debugging, where users can watch their variables change and see the script step through the original code. This feature may seem like a lot of unnecessary work, but it was the top request made by our designers at the end of the project.

"How'd You Do That?"

Implementing and maintaining the language was a significant task, and it took the work of several programmers to keep up with the designer demand for new COG library functions. But when compared to the time that would have been spent writing specific systems for things such as mission objectives, inventory management, powerup sequencing, puzzles, doors, elevators, and so on, using a language was a definite win.

After a period of uncertainty, most of the designers started becoming comfortable with the language and began to experiment with it. Strangely enough, the key here was trust. Once the designers were convinced that the language was safe enough that they wouldn't crash the entire game by writing a bad script, they began to try more interesting things with the language. After some time, they effectively took over the majority of the gameplay programming for JEDI, as planned. Some of the designers were so enthusiastic about the use of scripting that they later took evening classes in C and C++ programming, and at least one designer is moving into programming full-time. Even those designers who preferred not to work closely with the language found it easy to place and link existing scripts written by other designers into their own levels.

Perhaps the most important effect the language had on JEDI is illustrated by an example. One day, a large group was gathered around the desk of one of the level designers on the project. It seems he had created a puzzle script where the user hits a switch on the wall causing the water level in the room to rise slowly, carrying the player along to the top of the room. It seems simple, except we had just recently decided not to support moving water levels in the game. No one, including those who had worked on the engine and the language, could figure out how he did it.

I still don't know. I'm too lazy to find out. ■

Robert Huebner is a senior programmer at LucasArts Entertainment specializing in network and simulation programming. Prior to JEDI KNIGHT, he worked on DESCENT and other online titles for Interplay Productions. After JEDI KNIGHT, he will sleep for a month. He can be reached at virtual@lucasarts.com.

The Game of Ri k Ma ageme

Do you manage game development for a living? If so, ask yourself these three questions. What is your project's biggest risk right now? What are you doing to minimize that risk? And, what is your project's next milestone? If you can answer these questions quickly and concisely, congratulations. You are managing effectively and purposefully.

If you found yourself struggling to answer any of my questions, then I suspect that your development team has made one or more critical errors. How you answer these three questions is the best meter I know of to measure your efficacy as a project manager. How you answer could also be the best predictor of your development team's success.

In my experience, there are three lapses that cause game development teams to falter during the creation of a new game.

- The development team stops identifying, analyzing, and affirming the project's risks.
- The development team doesn't work intently to address and minimize the project's risks.
- The development team fails to regularly evaluate its performance and loses track of its progress against the projected development schedule.

What Is Your Biggest Risk?

If you can't name your project's biggest risk, then you do not have all of the information you need to lead your project effectively. Here's an example. During the development of Web.Max, a suite of Internet utilities, my team developed some very specialized technology to animate a sprite in the wallpaper of the Windows 95 desktop. The sprite, as designed, was going to be a friendly and playful "agent" that "lived" in your desktop and was always available to help you with your Internet tasks. The technology required to implement the desktop sprite was very complicated (it was intimately tied to the undocumented internals of the Windows 95 desktop), and we were elated when it finally worked. Once the technology started working, development of a key fea-

ture proceeded at the fastest possible pace and with the greatest sense of urgency.

We soon discovered, however, that our desktop sprite technology wouldn't be usable on our target system configuration. Ultimately, we removed the desktop sprite — a core, pivotal feature of the product as originally envisioned — from the shipping version of Web.Max.

What went wrong? My team (and that includes myself, as well) knew that we might not ever get a sprite to animate in the desktop. However, when the technology started working, we used it earnestly and extensively. We knew it would require "tweaking" as more and more users tested the software, but felt that the optimizations would be achievable. In the end — almost literally — we realized that no amount of tweaks would salvage the technology.

At some point, we thought we understood our new technology and focused our attention elsewhere. Because we didn't recognize and affirm that the technology was still a big risk, we didn't act to minimize its effect on the project and product. We mismanaged our risks and ultimately paid for our mistake.

As a development manager — whether you're a technical director, a creative director, a director, or a producer — you and your team must recognize and enumerate everything that is a risk to the project at all times.

Are You Minimizing Risk?

In my previous example, my development team dismissed the risk associated with our desktop sprite technology. That was our second mistake. Our first mistake was in not qualifying our technology initially and assessing its risk to the product. With better information earlier, we would have had the data and time to explore options.

Consider how my team managed the development, testing, and release of a new Windows 95 sound mixer, which we used for the first time in YOU DON'T KNOW JACK MOVIES. Until the release of YOU DON'T KNOW JACK MOVIES, my development team had been using the Windows 95 sound mixer that was created for the original YOU DON'T KNOW JACK title in October 1995. While that sound mixer was very good, it sometimes played stuttered sounds on PCs with slow SCSI cards and required the game's programmers to predict periods of CD-ROM drive inactivity. That mixer was also not extensively documented nor understood. In an effort to fix our mixer problems, the mixer was dissected and rewritten to better leverage the capabilities of Windows 95.

The new sound mixer was completed in early January, 1997. YOU DON'T KNOW JACK MOVIES was scheduled to be released just three short months later on March 21, 1997. The team had to make a decision: Should we ship the old system or the new system? The old system was proven; the new system held great promise and would improve the game play experience for almost all of our customers. What to do?

Our first task was to assess our risk. The author of the new mixer tested it extensively before releasing it to software quality assurance. In one test that he ran, the new sound mixer ran an entire week before he pulled the plug on the test. Next, the new mixer was incorporated into a special build of a shipping version of YOU DON'T KNOW JACK, and that build was tested internally and externally by more than 100 volunteer testers.

No problems were reported. The new mixer was a vast improvement over the old mixer, and we saw no risk to the MOVIES project. My team decided to adopt the new mixer, with excellent results.

New technology is only one of a countless number of risks. Do any of these other risks sound familiar to you?

"Joe Programmer has worked for two days on that bug and still has no clue how to fix it. What are we going to do?"

"Jane Artist is going to be in the hospital for the next month. Who'll do her work?"

"We're about three weeks behind schedule. Joe Sales says that the ship date can't change or we'll lose $100K of promotional expenses. What are we going to do?"

"I am having a hard time concentrating. Things are rough at home right now..."

"We don't know if we can get it to be that fast. Can we up the minimum system requirements?"

"I refuse to work with Joe. He's an idiot."

Killer bugs, staff departures, schedule delays, a team member's personal problems, tuning, personality conflicts, equipment failures, competition, and changing market conditions all add risk to your project. And no matter how well-prepared or well-informed you are, you'll always face unexpected problems that add risk to your project.

In fact, risk comes in so many forms and is so inherent in developing games that I now rank risk management as the most important task I perform as a development manager.

HOW MUCH DO YOU KNOW ABOUT YOUR PROJECT'S CURRENT STATUS? IF YOU CAN'T ANSWER THREE IMPORTANT QUESTIONS, YOU DON'T KNOW JACK. *by Martin Streicher*

How you (with your team) manage and minimize risk is as important to the success of your project as identifying and affirming that the risks exist.

What's Your Next Milestone?

Do you know what your team's next milestone is? Do you have a plan for achieving that milestone?

I'd be very surprised if you failed to answer "yes" to both questions. So let me qualify the questions and ask again. Is your team making satisfactory progress toward that next milestone? Is there anything jeopardizing your team's ability to achieve your next milestone? And, what are you doing to minimize those risks?

If you could not answer all of those questions, then it's likely that your schedule will slip. If you can't realize your short-term schedule, you don't have a chance of shipping on time.

I am a big proponent of "micro-milestones." Micro-milestones (a milestone is a clearly-defined goal and a deadline) force your team to achieve clearly defined and tangible results in one to three weeks.

For example, when my development team works on a new version of YOU DON'T KNOW JACK, the artists, writers, musicians, and developers work in concert to complete each feature one at a time, in its entirety, one after another. Each micro-milestone should be achieved in two to three weeks; there is one micro-milestone for each feature in the game.

With this system I am able to measure team and individual performance very accurately. I can see schedule slips in the same week that they occur and can react immediately to minimize risk.

In the past two years or so, I have abandoned traditional software scheduling and development management in favor of micro-milestones. Micro-milestones afford me much greater visibility of the schedule and of project risks. Achieving small milestones also adds to the credibility, morale, and confidence of the entire team.

And if you can realize your short-term schedule over and over again, you have an excellent chance of shipping on time. I am proud to say that the YOU DON'T KNOW JACK development teams have a flawless record of shipping products on time.

Carpe Diem!

I don't think game development teams intentionally make mistakes. I don't think development managers intentionally lapse in their leadership. I think development teams work very, very hard to realize their products.

And there's the catch. Individuals and teams often work so hard that they make mistakes that they wouldn't otherwise make.

Most of the mistakes my own development teams made were avoidable. The mistakes happened because I forgot to keep answering these three all-important questions. What is my project's biggest risk right now? What am I doing to minimize that risk? What is our next milestone?

QUESTION REALITY. Question the answers. Spend considerable time with all of the project leaders. Ask inquisitive and detailed questions. Review progress regularly and identify slips early. Verify the facts. Identify and challenge your team's assumptions. Audit estimates of time and expense. Ask for second or third opinions from the experts. Assess and re-assess risks with the leaders on your team.

ADHERE TO RULES EVEN IF YOU BREAK THEM. In general, if you must break rules, have rules to keep. It's extremely dangerous when a development team breaks rules and doesn't affirm the consequences. It's also dangerous to break rules and not change your behaviors and processes. If you do break your rules, follow these rules:

- Consciously and publicly state what rules you've broken.
- Identify the consequences of breaking the rules.

You must set time aside to think. Remove yourself from tactics, paperwork, e-mail, voice mail, spreadsheets, and meetings and just think

If you are a development manager, it's your job to anticipate mistakes and prevent them from happening. It's your job to answer my three questions every day. Here are ten suggestions to help you perform your job.

DON'T GET OVERWHELMED. Without exception, I've made more serious mistakes when I've been overwhelmed. Why? Because I allowed my guiding principles, judgment, and behaviors to be negatively affected by circumstances. When overwhelmed, I stop thinking. I lose my perspective on the project-at-large and focus too narrowly on a few priorities. I react instead of plan.

BE THE CENTER OF YOUR PROJECT TEAM. As the center of the project, you must serve as the clearinghouse for information. Progress, issues, problems, obstacles, and ideas should flow through you. When you receive new information from within the team, assess it and deliver it to someone who can use it. If you receive information that affects the entire project, broadcast it. Match answers to questions.

- Design and implement a plan to counter any negative consequences and minimize the impact on the project.
- Immediately create a new set of rules and publicly state the new rules.

Don't break the new rules. If you do, follow these rules.

THINK. You must set time aside to think. Remove yourself from tactics, paperwork, e-mail, voice mail, spreadsheets, and meetings and just think. Think about your risks. Is each risk assigned to someone to resolve? When will each risk be resolved? Is there a risk that is languishing? Is it languishing due to lack of attention, or is it being intentionally ignored? Think about the people with whom you work. Are individuals rested? Does anyone need a break? Can you relieve some stress or anxiety?

REVIEW THE STATE OF YOUR TEAM. Estimate, measure, estimate, and measure in an endless cycle. Track progress closely. Ask your team members if they believe in their individual schedules and in the schedule of the entire project.

Explicitly state who is doing what. Are there obstacles in anyone's way? Will there be any soon? Is there anything you can do to facilitate progress? Is everyone performing adequately? Do you need to take any corrective action?

DON'T FIRE "YIN" OR "YANG." A healthy development team has tensions. Tension exists between engineering and software quality assurance, between art and engineering, and even between the development team and outsiders (supervising management, executives, even other teams). Tension is natural in product development and is evidence that each part of the team is exhibiting and practicing its own form of ownership. Encourage investment and ownership. Encourage differing opinions. Allow each part of the team to challenge others. Build and maintain a healthy system of checks and balances. Of course, don't let any one "Yin-Yang" relationship get too tense or take the team out of balance.

Focus on and eliminate your biggest risk over and over again. Remove the next obstacle to your project before you reach it. When a new risk crops up, announce it, and assign someone to resolve it. Identify alternatives, tradeoffs, and possible solutions.

EVALUATE YOURSELF. Pay attention to your own performance. Are you suffering from manager's "tunnel vision" — focusing too narrowly on a few tasks while others go unmanaged? Are you thinking? Are you doing? If you have a supervisor, imagine how they'd evaluate your performance. Whether you think in your car on the way to work or in the shower when shampooing your hair, take time to consider what is happening in your project.

LEAD BY EXAMPLE. You must be an excellent manager. Communicate. Explain. Encourage collaboration. Respect others. Recognize achievements and reward appropriately. Be responsible. Be honest. Keep your word. Challenge others. Demand excellence and professionalism.

Practice Your Mantras.

Do you recall the three questions I initially presented? Do you feel better prepared to answer those questions now? Try again.

- What is your biggest risk right now?
- What are you doing to minimize that risk?
- What is your project's next milestone?

So, how did you do? ■

Martin Streicher is a producer at Berkeley Systems in Berkeley, Calif. He graduated from Purdue University in 1986 with a Bachelor's and Master's degree in Computer Science and has been a software development manager since 1989. Martin's entertainment software credits include the Disney Screen Saver for Windows, After Dark 3.0 for Windows, You Don't Know Jack v1.0, and four other versions of the popular CD-ROM trivia game. He is currently writing, producing, and directing a prototype of a new game that he created.

Whe Mo io Ca e Bea Ke f ami g

otion capture is fast becoming a “darling” technology in the game development world. This may be attributed to the maturation of the hardware and software tools. It may also be the newfound enthusiasm of project managers and producers who are embracing “mocap” within the various video game and visual effects industries. For whatever reasons though, motion capture has become a favorite among many game professionals as they strive for more realistic movement within the 3D worlds that they create. If you potentially have a need for this technology or are simply interested in an overview of the issues concerning this process, read on.

Beginning, Middle, End

Motion capture is still pretty new on the evolutionary tree of technology. Too many people simply haven’t been intimate with the entire throughput process, and this is where potential trouble can begin. Data has to be extensively manipulated before to get decent results. From acquiring the data, to transforming that data into a certain file format, to building specific models and finally pointweighting and attaching the mesh, motion capture is a process. Thus, the decision for or against using motion capture should be made during the preproduction phase of a project. We’ll say up front that we’re strong believers in “detailed” preproduction.

Certainly, when planning to use motion capture for game development, a number of questions come to mind. For instance, in a level-based fighting game, you might ask yourself: How many moves there are in each level? Can those moves be done with traditional keyframe techniques? Which process will get the job done more efficiently? What’s my budget going to allow? Do we have someone in house that has worked with pointweighted meshes before? Everyone thinks that they’ve already considered these issues — but have they really?

Common sense still prevails. Be prepared with a set of plans that are on paper before you head to the studio. Most teams come in with a spreadsheet that serves as a breakdown of the shots by level. Each spreadsheet is different, yet they all have a few common columns. Take Number, Actor, Level, Action, and so on. Creating these spreadsheets is a good way to get the ball rolling; the project should be taken further with either story boards or an overhead view diagram of the action to take place. (Diagramming an overhead view, scene by scene, works well. Yet very few people do this to help plan their shoots.) “Floorplans” are useful when your desired move takes place over a larger space than the mocap volume will allow; a couple of different takes must be “blended” together. The issue of “long” moves comes up quite

often in game development; this qualifies it as a good reason to use motion capture. A complicated move that physically covers a lot of terrain suits motion capture capabilities well, but it still helps to have the more complicated moves diagrammed to the exact step (Figure 1).

As the game industry has adopted some of Hollywood's ancillary habits, it is also picking up some good habits as well. The trappings of a film-style shoot can be involved however, so be sure to include video reference, wardrobe, stunt support, and craft services on your check list before heading to the studio. The bottom line is to create a productive environment from which to garner the best movement possible from your actor or performer.

Motion capture "shoots" are done in a similar manner to the film industry shoots. Story boards are drawn up, actors are positioned at their marks, a director calls for "Action!" and the operator of the motion capture station rolls tape... er, hard drive. After numerous takes are recorded, the director and/or game production managers determine which are the best takes, and then the mocap crew begins to clean up the data. After the data is "massaged," the files are converted into a skeletel format that is ready for "attachment" to the meshes or models. This is where the skin meets the bone.

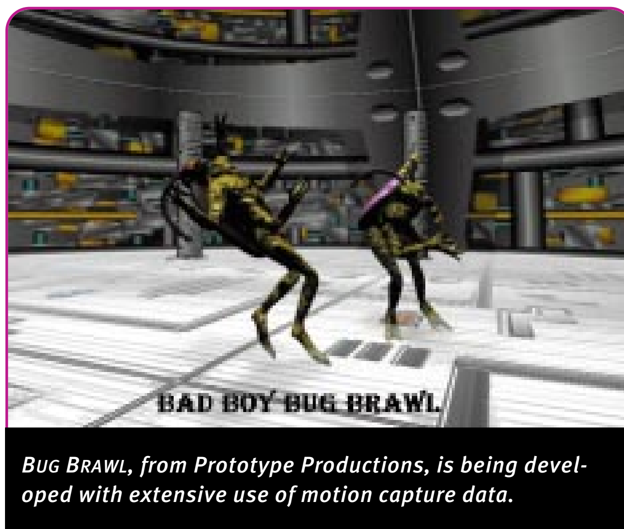
Costs and Comparisons: Traditional Keyframe vs. Motion Capture

We don't want to start any more controversy on an already dead issue. Motion capture has strengths and weakness just as anything else that is new and on the edge of the technological world in which we live.

The gaming industry has seen improvements in hardware and software implementation over the past couple of years. New games have a more realistic look and feel that wasn't possible to create during the 16-bit era. Along with an increase in CPU horsepower comes an increase in demand for more realistic-looking characters, as well as more fantastic creatures that inhabit these virtual environments. Motion capture fits into this scenario by providing data sets that mirror humanistic, or bipedal movement.

Humans are typically very difficult to animate using traditional keyframe animation techniques, especially when trying to depict complicated, two-person moves such as a sword fight. Some very talented animators are capable of humanistic, bipedal animation, but it's difficult to key an entire sword fight sequence with interactive clashing of weapons and all.

There are numerous ways to skin a cat, so determining whether or not motion capture will be used for a project depends, to a large degree, on the available in-house resources. Typically, the amount of motion data used for a real-time game is contingent on a number of factors. These factors are directly related to the different stages of the throughput pipeline. From scripts and story boards to skin and

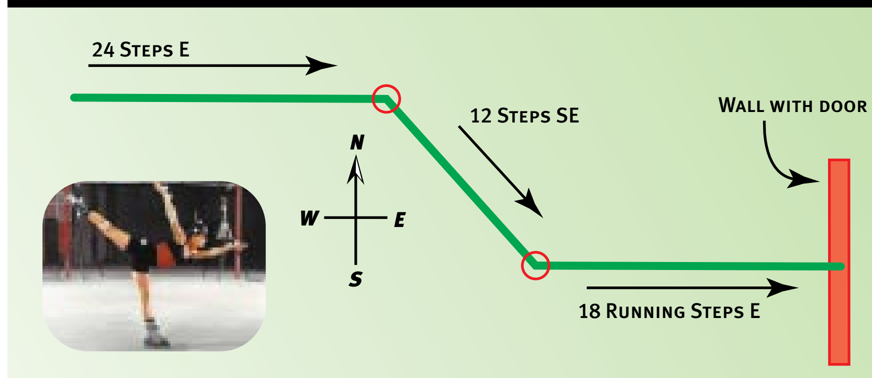


BUG BRAWL, from Prototype Productions, is being developed with extensive use of motion capture data.

WHEN GOING FOR NATURAL-LOOKING CHARACTER ANIMATION, MOTION CAPTURE TYPICALLY BEATS OUT KEYFRAMING... FOR A PRICE.

by Damon Knight and Tom Tolles

FIGURE 1. *It might be a good idea to draw a diagram of complicated moves.*



A series of screen shots from Activision's APOCALYPSE. *The animations were recorded at House of Moves during a motion capture session with motion picture star Bruce Willis.*

bones, any one of these stages can be performed with great success or whittled at with disappointment. A game development company should determine its own role in making the motion capture route work. Does the company employ staff that can do the attachment? Can that staff also clean up data? Does the company have the necessary contacts to sub out the attachment work? Or should the animations just be keyframed?

A lot of companies have strong in-house animation staff; this is a huge bonus. In this case, it makes sense to go with your in-house strength. However, as one project manager related, "It may be that I have a great animator, but having a great animator that can also play the sport that he is working on for the game is pushing the odds a bit." The next question then becomes, is it cost effective? For the company that may be smaller or has certain other in-house strengths, motion capture is a good way to go.

Keyframing is a valid and sound choice for game development; it is presently the standard for animation. However, time changes everything. The progression of the software and hardware makes for a promising future for mocap.

Performance Animation

The nice thing about using a performer to capture movement is that an actor can be given direction. The human brain interprets a director's requests very effectively. Interpreting a director's instructions is best done by a human, as they are much faster than a computer at "grokking" the desire of the director — the last time I checked anyway. Working at the speed of human comprehension, one can usually amass between 80 and 100 takes a day, depending on the complexity and duration of the movements required.

Blending the Flesh and Attaching the Skin

There are a number of issues that go along with the implementation of motion capture data to 3D characters. Point weighting and attachment are favorites with most 3D artists. The polygonal characters that are to be attached to the mocap data may have to be created in a more specific manner than usual (as in, more polygons at the elbows and knees for maximum flexibility). It's also a good idea to know the accurate measurements of the performer from whom the data was acquired. The object here is to create the mesh around the initial data's position or "daVinci pose" that is imported into frame 0. This assures that the mesh will be exactly proportional to the bones, which is helpful throughout the process. Having a well-built and proportionally correct model is a key element.

When the data is imported into the software package, it looks like a skeleton. The mesh is then imported and the vertices are set to specific links.

Actors can be given direction. Interpreting a director's instructions is best done by a human, as they are much faster than a computer at "grokking" the desire of the director.

This method of skin attachment is commonly called point weighting or attachment. The skin, or mesh, may be either a single or jointed skin, depending on the software and required use. It is wise to do attachment right away, so as to get a head start on any problems that the actual motion data may impose on the newly attached skeleton. The biggest problems come when the moves are really aggressive and the mesh has to be stretched and twisted into radical positions. Some tearing and deformation may occur. Once you get the mesh attached to the skeleton though, it's possible to import different motion data sets — this is where you start to make time. You can then boast to your friends that you did ten moves in a single day. Your project manager or producer will be very happy at this point. (At least until they tell you that all those moves need to link together.)

The second major issue concerns "blending." This is when many small moves must be capable of seamlessly ble nding into each other as the player presses certain combinations of buttons that correspond to the moves described in the game requirements. This technique is done by stringing all of the takes together to make one long complicated move that can then be accessed accurately and quickly by jumping to particular frames…er, keyframes. Blending is sometimes difficult, but can be helped along by various plug-ins or a good animator that digs in and creates a workaround. Dave Matthews from Konami says, "The plain fact of the matter is that you do have to work at it. Motion capture data works well but it isn't magic. [All one needs is] just hard work and an understanding of the limits of one's resources."

You Still Have to Buy the Software!

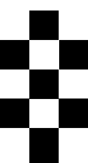
There are a limited number of software packages that implement motion capture data well. Softimage and Alias are obvious winners in this

arena; however, there are a number of plug-ins for 3D Studio MAX that are available to developers. Oxford Metrics and BioVision both have new plug-ins, and Kinetix is sure to do something in the near future with all the success that MAX is enjoying. The 3D MAX package is unique in that we can implement motion capture data into a PC-based piece of code and actually get stunning results. Talk about stellar results on a cost effective platform.

Having used the Oxford Metrics plug-in for MAX, we can honestly say that it does a good job in conjunction with Physique. This little piece of code has key-reduction capabilities to reduce unwanted data and keep files from ballooning. It also has features that allow you to import specified segments of the file, and it lets you blend moves together with a bit of help from the code itself. This software combination actually loads quickly and lets you do a number of things that only time and experience will reveal, such as perfect run cycles. BioVision has a MAX plug-in also, called Motion Manager, and it too has a rich feature set. The BioVision file format is easy to work with, which is a blessing when trying to rotate bones into certain positions for attachment.

Alias and Softimage are excellent packages, as we all know, but not everybody uses the UNIX platform of choice for game development.

Mocap may not be right for every project, but it seems to be hard to beat for capturing the nuances of the human movement.



Going Nowhere and Everywhere

It's important to point out that mocap capabilities will only get better with time and will eventually evolve into an integral part of synthetic creations where bipedal creatures are present. It's not for every project, but it seems to be hard to beat for capturing the nuances of the human movement. ■

Damon Knight has been working at House of Moves motion capture facility for the past couple of years as an all-around data twister. He has a graduate degree in Film and is an old-time computer user (his first was an Osborne back in 1981). He is currently developing games with his new company Prototype Productions. You can contact him via e-mail at damonK@moves.com

Tom Tolles graduated from Stanford with a degree in Mechanical Engineering and received his MBA from UCLA. He is the owner of House of Moves and has twelve years of experience in 3D animation. You can contact him at tomt@moves.com.

Im leme i g Mi ed Re de i g

The growing popularity of PC 3D hardware has sparked debate among software developers as to whether to continue developing software rendering engines or to abandon them completely in favor of hardware. Both choices offer advantages and disadvantages. Luckily, there is a way to get the best of both worlds: combine, or mix, the two rendering methods. By using mixed rendering, gamers and programmers get the flexibility of software and the speed of dedicated 3D hardware. This happy medium can exist within the standard graphics APIs for Windows: DirectDraw and Direct3D. Several methods exist for mixing software and hardware rendering — color keying, overlays, alpha BLT, Z-buffered BLT, or animated texturing. We have implemented samples using Direct3D on several hardware accelerators. Performance issues arise in synchronizing hardware and software, but these problems are solvable.

Mixed Emotions? Mixed Rendering

Game developers want to differentiate their titles as much as possible, in visual quality, speed, and complexity. The influx of 3D accelerators in the mainstream market raises potential 3D quality and performance, while leveling the visual complexity playing field. You can now freely exploit hardware acceleration. However, you lose the fine-tuned control of the visual quality when hardware (via an API) performs all of the rendering. The API of choice for most Windows game developers, Microsoft's Direct3D, provides hardware-independent 3D acceleration; but it can also limit uniqueness of the game's look.

For example, you may want the great features of your own customized 3D software engine. You may also want to push the limits of quality and visual complexity with techniques commonly used in offline rendering engines (for example procedural textures, curved surfaces, and ray tracing). However, you'll still want Direct3D and its Hardware Abstraction Layer (HAL) to render the scene as quickly as possible on widely installed 3D hardware.

Are you stuck with these mixed emotions towards Direct3D? Perhaps not. The speed and capabilities of 3D hardware and CPUs will increase significantly for the fore-

FIGURE 1. *Scene from Marble Bagels Traveling Through the Tunnel.*

seeable future. However, you won't see clever new techniques in "fixed function" 3D hardware accelerators until years after they have been implemented in software, if ever.

We illustrate the concept of mixed rendering using the familiar Direct3D sample application, "Tunnel." In our case, we'd like to think of the tunnel as our background as we send an object traveling through the tunnel. We'll use a procedurally textured torus (a marble bagel?) as our object.

"Marble Bagels Traveling in the Tunnel" might not sound like the most clever game in the world, but it does illustrate the required infrastructure of mixed rendering and provides an example of an advanced technique that isn't available in any 3D accelerator. It should also be stressed that there's nothing special here about using procedural textures. Pick any special rendering technique that you like and apply it within your scene.

TABLE 1. *Performance Results for Two Configurations (Gouraud shaded, texture mapped)*

	Hardware Only	Mixed Rendering
Configuration A	21 fps	25 fps
Configuration B	28 fps	27 fps

Config A: Pentium II, 233 MHz, with Creative Labs 3D Blaster PCI, with bilinear filtering

Config B: Pentium with MMX Technology, 150MHz, with Matrox Mystique, point sampled textures

BY DRAWING UPON HARDWARE *AND* SOFTWARE FOR YOUR RENDERING, YOU CAN ADD SPECIAL FEATURES TO YOUR GAME THAT OTHERWISE WOULDN'T BE POSSIBLE.

by Haim Barad & Mark Atkins

The tunnel has a low number of large polygons (think of it as our background), while the bagel is an object with a larger number of small polygons (representative of a complex foreground object). Figure 1 shows a sample screen from the application.

A Bit of History

Our first investigations into mixed rendering actually made concurrent use of both the Direct3D HAL and Hardware Emulation Layer (HEL). We combined the Direct3D Twist and Tunnel applications into a single application (Twist rendered in software and Tunnel rendered in hardware). This example could be rendered entirely with hardware, but using mixed rendering shows the methodology and allows comparisons of performance. (Note: While we don't advocate mixed rendering for techniques that are widely available in 3D accelerators, we use this example just to illustrate the concurrency available in some 3D hardware)

Performance depends on the 3D hardware, since different accelerators and their drivers allow for different amounts of concurrency. That is, better hardware accelerators permit the CPU to continue calculating while the hardware draws, while lower-end hardware tends to require the CPU to idle, waiting for hardware completion. Performance also depends on the rasterization load balancing the application achieves for the software and hardware. This application's performance using the multithreaded methodology described later appears in Table 1.

The performance in this example is not an apples-to-apples comparison, as the 3D Blaster supports bilinear filtering, while the Mystique does not. In addition, the different platforms that they were on also complicate comparisons. The comparison should be made separately between the "Hardware Only" and the "Mixed Rendering" columns for each configuration.

The results in Table 1 show that the performance of Configuration A improved when we mixed software and hardware rendering, while the performance of Configuration B actually degraded a bit. This improvement can be traced to two key factors: rendering and hardware concurrency.

When and Why It Works

Originally, our rendering process contained a single thread, which fed triangles to the 3D accelerator card as fast as possible. If an accelerator card is fast, the thread runs at the CPU's maximum rate. Unfortunately, there are situations and cards that process data more slowly, or times when data needs to be blocked entirely. If the card's command memory is full and it cannot accept any more triangles, or if it needs to wait for the vertical blank interval (VBI) to flip the drawing surfaces, the thread has to block (wait).

By adding a second thread, the rendering process has the opportunity to do something useful in case the original hardware rendering thread blocks. In this case, the second thread is software-only rendering. This concurrency between the CPU and the accelerator accounts for the performance increase in the 3D Blaster — we had two rendering engines at work simultaneously, resulting in a performance gain in spite of the overhead required to composite the two images. While the Mystique didn't support concurrency (nor bilinear filtering) and resulted in a small perfor-

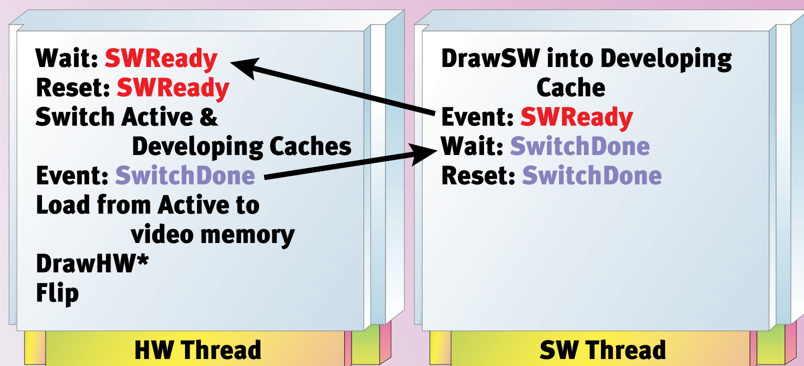


FIGURE 2A. *BLT composite and single buffer in software thread.*

mance loss, future 3D hardware and drivers should have better support for concurrency.

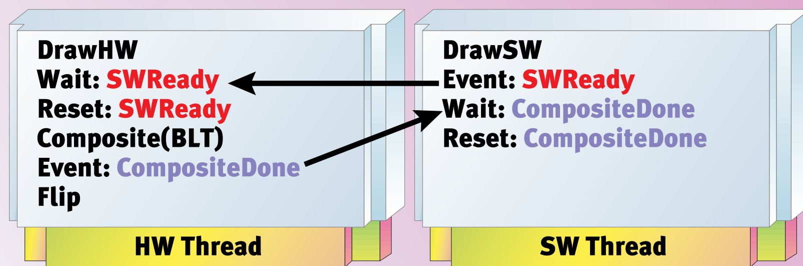
The motivation for mixed rendering isn't performance. The ability to integrate special features and advanced techniques is the overriding benefit. To illustrate this, let's first look at the multithreading infrastructure.

The Multithreading Methodology

Mixed rendering should be multithreaded to exploit concurrency and must manage the priorities of the two threads during different stages of the rendering process. Figure 2 illustrates a software methodology for two cases of multithreaded mixed rendering without deadlock: one case using a separate BLT composite and a single buffer in the software thread, and another case using a texture mapping composite and a double buffer in the software thread. Both scenarios assume a complex flipping surface (that is, backbuffer and frontbuffer) for the hardware thread.

In Figure 2a, the hardware thread handles the drawing of the parts of the scene done with 3D hardware, as well as the composite/flip operations. The software rendering is done within a separate thread to an offscreen buffer in system memory within a loop using synchronization events. Two distinct synchronization events (`SWReady` and `CompositeDone`) are used to avoid deadlock in case the rasterization load becomes unbalanced (that is, much heavier in one thread than the other). The `SWReady` event is set when the software thread has finished its frame, and its results are ready for compositing (using a BLT operation) with the frame generated in the main thread. The software thread must then

FIGURE 2B. *BLT composite and single buffer in software thread.*



wait for the results of this rendering to be consumed by the main thread (via BLT composite), and then signaled by the event `CompositeDone`. Once the composite has completed, the software thread can begin working on the next frame.

Using multiple buffers (Figure 2b) on the software thread can allow progress to continue into the next frame(s) without waiting for the results to be consumed from the current buffer. We name the two buffers in the software thread "Active Cache" and "Developing Cache." The reason for these names will become clear later, when we discuss a way of integrating "image caching" into the mixed rendering methodology. In this case, the software thread always draws into the developing cache. In the hardware thread, we must wait for the software thread to get one frame ahead of the hardware thread in order to provide a ready-to-use texture for compositing into the hardware-rendered scene. The load from active cache to video memory is done for each frame into the hardware's video memory. Then the hardware draws the entire scene ("`DrawHW*`"), which includes the texture map composite, onto a single polygon in the hardware-rendered scene. We'll discuss this compositing method later.

You must also make a tradeoff concerning memory allocation (extra buffers) versus increased performance when considering these alternatives.

Compositing the Two Scenes

Many methods exist for compositing the results of the hardware and software threads. DirectDraw provides several options for the BLT operation.

Z COMPOSITE WITHOUT Z-BUFFER? In spite of the fact that BLT with Z isn't currently enabled in DirectDraw, we can composite distinct (as opposed to interpenetrating) objects in the scene based on their Z values.

The results of the software rendering are texture mapped (transparently) onto a rectangle in the hardware backbuffer. We call this Single POlygon Textures (SPOTs); you can take advantage of the 3D hardware's capability for filtered texture mapping. This also allows the software rendering to be done at a smaller size and then stretched to tune performance or achieve special effects like distortion. Figure 3 is an example of a mostly hardware-accelerated background with a high-quality, albeit small software-rendered object. The hardware will texture map the motorcycle to the SPOT during the drawing of the hardware scene. This case also lets the software object (the motorcycle) actually interact with the hardware-rendered scene (such as go behind trees or obscure other hardware-rendered objects). All we have to do is move the SPOT to the proper place in the scene and transparently texture map to the SPOT.

Compositing via texture mapping also offers a performance advantage. Many 3D cards perform much better if you do not intersperse 2D operations (such as, BLT) with 3D rendering. This often causes a large performance hit. In the section on concurrency, you'll see that the BLT operations on hardware with good concurrency will cost a lot in performance.

Some hardware will even support differing pixel depths for the source and destination of the BLT or texture mapping. Also keep in mind that the performance of texturing out of system memory will be improved with AGP; higher bandwidths and the explicit loading of the texture to video memory won't be necessary.

Scene Partitioning Strategies

A number of guidelines will speed you down the road to mixed rendering. The CPU is required when rendering the objects in your scene that exploit specialized or advanced 3D techniques. Often these are interesting foreground objects. These objects are usually composed of smaller triangles, and the performance of software rasterization on small triangles is more competitive with hardware rasterization. In other words, the CPU is more competitive with triangle throughput than it is with pixel fill, and creating the commands and data for hardware requires a lot of time compared to the time it takes the hardware to draw small triangles. This strategy corresponds roughly to background/foreground object partitioning.

Again, mixed-mode rasterization shouldn't be used for performance reasons alone. Until the infrastructure for exploiting concurrency is enhanced, there may be no appreciable performance gain (and possibly a performance degradation). However, forming a consensus on a standard mixed rendering methodology will help the industry avoid roadblocks for better performance.

Procedural Textured Object Inside a Hardware Background

The application "Marble Bagels Traveling in the Tunnel" demonstrates an advanced technique not supported by 3D hardware, namely proce-

FIGURE 3. *Composite via texture map to SPOT.*

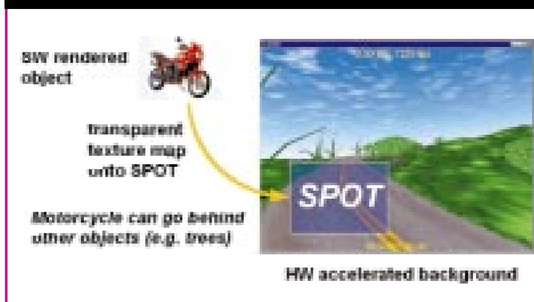
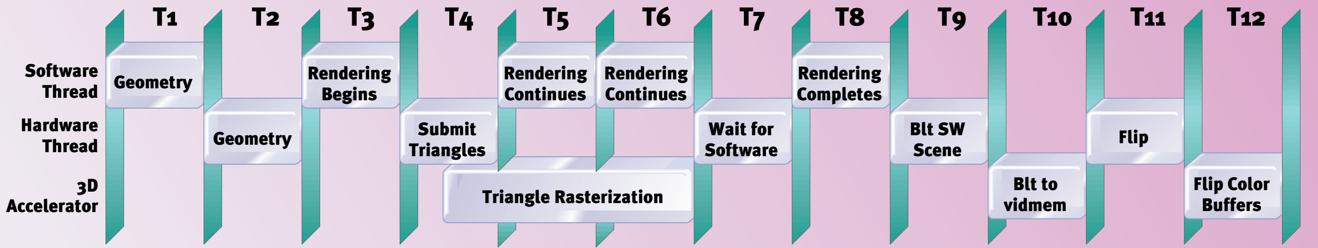


FIGURE 4. *Progress of both threads.*



dural textures. We made use of a Perlin Noise generator and turbulence (optimized for MMX technology) to produce the marble-like appearance. A discussion of procedural textures is beyond the scope of this article. However, we do include the complete code and executable for the mixed rendering example on the *Game Developer* web site.

Performance Issues

Running two separate threads is the first step toward squeezing extra performance out of the system. Like most multithreaded operating systems, the Windows thread scheduler dynamically changes thread priorities. Higher priorities indicate that the thread should be run more often; threads with lower priorities run less often. A thread that is waiting for an I/O operation to complete will have it's priority decreased. Since I/O operations typically take anywhere from tens to hundreds or thousands of CPU cycles, there's no reason to waste time checking on a thread that isn't ready to continue. However, when the scheduler determines that the thread has just completed an I/O operation, it's priority will be increased. This gives the thread a chance to process the result of its completed I/O operation.

Dedicating one thread to software rendering and another to hardware rendering leverages the behavior of the thread scheduler. The hardware rendering thread performs several I/O intensive operations — sending triangles to the hardware rasterizer, compositing the software rendered objects, and flipping the drawing surfaces. The scheduler decreases the priority of the thread as each I/O event occurs, and then raises the priority when it completes. Even though both the hardware and software rendering threads begin at the same priority, decreasing the hardware thread's priority allows the software thread to run more often. This extra CPU time allows the software thread to complete more rendering while the other thread waits for I/O completion.

Concurrency. Running separate threads for hardware and software rasterization is one form of concurrent operation. There is also a second level of concurrency taking place in the program — that between the hardware rasterization thread and the 3D hardware accelerator.

Ignoring other threads in the system, the scheduler switches between executing the software and hardware rendering threads. The hardware accelerator essentially gives us another thread of processing. Most of this thread's activity takes place on the hardware accelerator itself, giving us true parallel operation with the threads on the CPU.

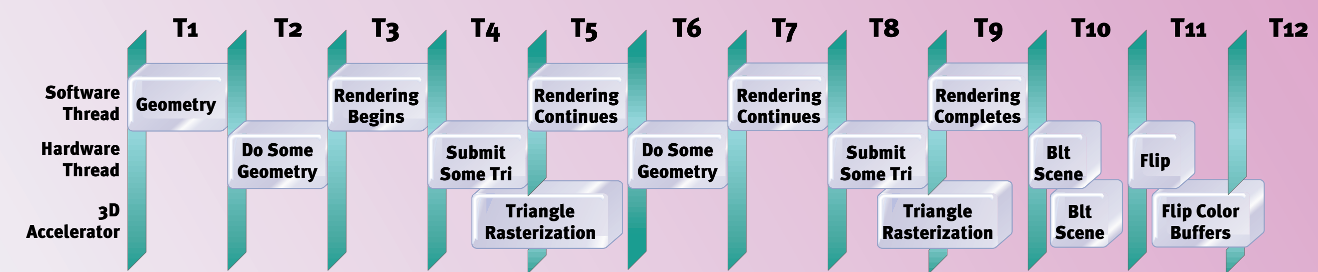
Simulating this parallel execution (Figure 4), demonstrates how the scheduler switches between the software and hardware rendering threads. At time T4, the hardware rendering thread submits triangles to the 3D accelerator. This causes the thread to block, pending the completion of triangle processing. The software rendering on the CPU and the triangle rasterization on the accelerator then run in true parallel execution.

One thing to note are the potentially long time periods when either the CPU or the 3D accelerator aren't performing any useful operations. This is due either to idling or stalling. The 3D accelerator sits idle, waiting for something to process from T1 until T4. The CPU is stalled at T9 and T11 waiting for the BLT and flip operations to complete before continuing.

By slightly recoding, stalls and idles can be reduced and the entire operation can take less time (Figure 5).

The first modification is to control the number of triangles submitted to the 3D accelerator. For any API or dri-

FIGURE 5. *Progress of both threads after recoding to reduce stalls and idles.*



ver, processing data has an associated overhead. In the case of Direct3D, approximately 100 vertices worth of triangles is the minimum to submit to achieve the best performance. This performance is constant, regardless of whether you're using the **ExecuteBuffer** or **DrawIndexedPrimitive** functionality. Any less than that, and it will still take about the same amount of time to complete and return due to overhead.

On the other hand, thousands of vertices shouldn't be submitted at once, either. Submitting vertices for rasterization should be arranged in such a way as not to idle or stall the CPU or the 3D accelerator for long periods of time. Breaking a large number of vertices into multiple submissions, T4 and T8, allows better throughput and parallel execution.

The second modification is to remove stalls in the hardware rendering thread. The two biggest culprits here are waiting for the BLT and the flip operations to complete. By running these asynchronously, the hardware rasterization thread no longer needs to block for a long period. The 3D accelerator can also queue and perform the operations in parallel with the beginning of the next frame.

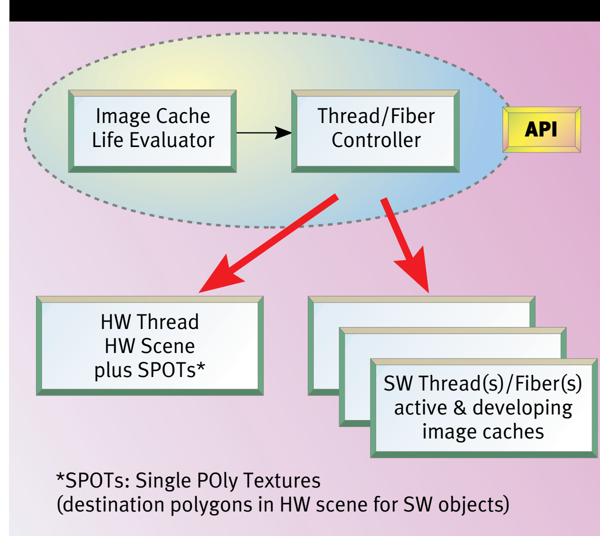
MR² (Mixed Rate, Mixed Rendering)

Because we're already rendering the scene by compositing layers, we can also consider a way to handle rendering threads that are at different frame rates (and even asynchronous). Specifically, let's assume that we want to perform some very advanced and intensive software rendering that will be slower than the hardware-rendered scene. We can make use of image-caching techniques to achieve this. This is a technique adopted by Microsoft's Talisman initiative (see 1996 Siggraph paper by Shade et al. entitled "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments").

Let's illustrate this with an example. Suppose the frame rate of the hardware-rendered portion of the scene will be around 60fps on a specific platform. Also assume that the software-rendered portion of the scene is achieving around 12fps. We have a 5:1 ratio in frame rates. Further assume that the objects rendered by the software thread don't change too much from scene to scene. We then can make use of the rendered scene as an image cache to be used for the next four frames that will be warped properly. The 2D warping function is much less costly then rerendering the scene (and supported by some hardware) and will allow the software thread to continue working on the next frame.

42

FIGURE 6. *MR² architecture.*



Of course, we'll need some infrastructure to support this. Figure 6 illustrates the architecture for MR². We see an opportunity for an API to control the basic process. For example, we need a controller to estimate image cache life. The life is the number of frames for which it can be used in addition to the original rendering. In our example, the life is five.

The software renderings can be broken into several threads, depending upon scene complexity and requirements for each of the software renderings. For example, you might have one object that uses a rendering technique much slower than that of another. These two objects can operate as independent threads. The results of all the software threads are then composited by texture mapping (with transparency) onto SPOTs.

Each software thread should make use of a double-buffering method. You now understand from where the names "developing cache" and "active cache" come. The developing cache is analogous to the backbuffer, as it is the surface to which the software thread is currently rendering. The active cache is analogous to the frontbuffer, as it is the surface that the hardware thread uses for compositing (warp and texture map).

MR² as applied to mixed rendering is an idea only in it's infancy. But we expect that formalization of a mixed rendering methodology will help us get a better handle on its requirements and performance characteristics.

Making It Practical

The idea of mixed rendering is not new. As soon as there were 3D accelerators, there was already the desire to mix software and hardware rendering techniques. Our goal is to develop a workable methodology for use with APIs, so that the industry can address the needs of mixed rendering. Some examples of issues already being addressed are

- BLT with Z (not yet implemented in DirectX)
- better concurrency and synchronization in the API and its drivers
- and development of "intelligent schemes" for control (for example, MR²).

It is our hope that further development of methodologies for mixed rendering will provide a valuable alternative path for game developers to exploit both the raw performance of 3D accelerators as well as the flexibility of software. The benefit will be enhanced quality and performance of 3D games. ■

Haim Barad is a staff engineer and technical leader of Intel Israel Software Lab's 3D Team. He received his BSEE and BSCS from Tulane University and his MSEE and Ph.D. from the University of Southern California. He can be reached at barad@iil.intel.com.

Mark Atkins is an applications engineer at Intel, specializing in 3D graphics software and CPUs. He received BSEE and MSEE degrees from Purdue University. He lives to bicycle. You can contact him via e-mail at Mark_Atkins@ccm.sc.intel.com.

Po e o he Kid !

The mind of the child lies at the vortex of software design. The major breakthroughs in user interface, such as icons and multiple application windows, originate in work done for children. So does the concept of the laptop. But alas, the educators have grabbed the market of childhood software and perverted it into the polar opposite of everything for which the personal computer stands. Rather than empowering the children and liberating their inner creative selves, they intimidate and enslave the child to the machine. Rather than teaching the very first lesson every child of the modern techno-world must learn, that Man controls the Machine, they subliminally inculcate the child with the notion that Man must obey the Machine or fail.

But there is hope: Game developers, with their unbridled, insane imagination and nonconforming nature, are the ones who can crawl into the child's world and liberate those latent inner powers that have remained dormant. Here are some game design strategies for children up to seven years old.

Know Your Market

Somewhere between 32 and 36 months, a child can learn to manage a mouse. A four year old can easily be a master of the machine.

However, the higher end, seven year olds don't yet get along with sophisticated game-controllers. This is one of the reasons none of the console makers have had any success to date with this age group. Another reason is that this age still identifies with childhood and, frankly, thinks very differently than older children (see "A Case for Inappropriateness"). Around seven or eight, a major reorganization of grey matter takes place,. By nine, you have everything it takes to make a full-blown Ultra-64 junkie. These kids no longer want "kid's stuff" — they want the real thing. So let's try to understand what goes on when children age one to seven hit silicon.

AN ADULT WILL PURCHASE AND SHARE THE PRODUCT. A child typically doesn't even choose the software. It's usually picked by a parent or grandparent according to what they think the child will enjoy. They almost always mitigate the guilt of spending so much for a kid's toy by justifying it as "educational."

All this means that your software must have the following qualities:

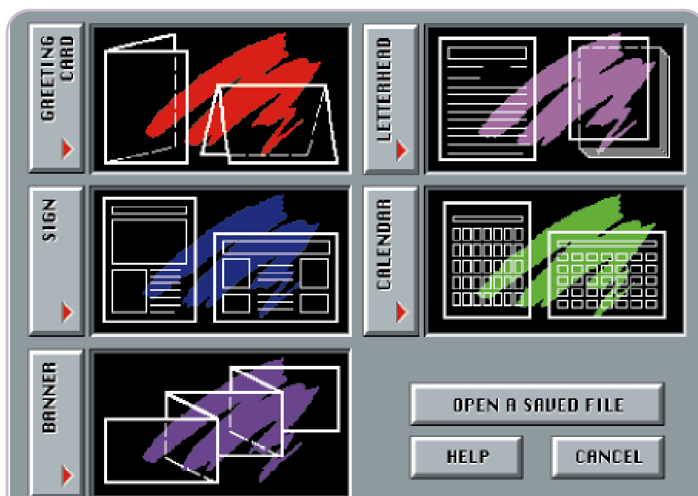
- It should be enjoyable and attractive both for a child and an adult.
- It should allow plenty of opportunity for the child and the adult to interact with each other. On the other hand, the child should be able to figure out how to have fun without any adult supervision. (This is what really takes brains.)
- It should be versatile and deep enough to be fun to a wide age-group of children.
- It should have qualities that grandparents will consider educational. (You and I know that if it's good software, it's educational. But it might be hard convincing Grandma and Grandpa of that.)
- It mustn't have anything that could make adults uneasy about recommending it for small children. In other words, avoid violence.

Know the child. Not just you, but the artist, the animator, the programmer, the sound and music people — everyone creatively involved in the project must have a real, first-hand, in-depth feel for the child.

The problem is that everyone feels that they're already an expert on children. After all, no one has managed to escape the first-hand experience of being one. Still, few of us have maintained a faithful recollection of what childhood is really all about.

I recommend watching a child who's at a computer. There's no more lucid window into the child's mind. They talk to themselves, to others, and to the machine more articulately than at any other activity.

Try talking with a child as he or she sits at the machine to get even more



The Print Shop (Broderbund) obstructs many children younger than 10 from thinking creatively or working independently by providing mind-boggling abstractions and procedures.

Developing a Successful Children's Game requires careful design and a thorough understanding of the pre-adolescent mind.

by Tzvi Freeman

feedback. Be careful not to influence their decisions or processes. You want the raw child — not one tainted by your external influence.

A vital connection to the game you're producing is essential. Children's software isn't the sort of thing you contract to out-of-house generic programmers. People that make children's software don't "do that also" — they are people that specialize in children's software.

Know what the child needs. Children do a good job of looking as if they're wasting time, but secretly they are in the business of educating themselves about how the world works. That's how they have fun. They just need you to facilitate their discovery by providing them with the right tools and environment in which to explore.

In other words, they need you to empower them. Just as you empower the adult, the teen, or the older child to build cities, blow up monsters, or fly jet aircraft, so you empower the younger child to explore and discover a world to which they can lend some sense.

The tools the child needs to have to do this have two primary qualities: versatility and comfort. Versatility allows the child the freedom and power to explore. Comfort allows the child the confidence to go for it. When you think about it, these are the same two qualities that we look for in a good car, home, computer, or any other adult tool. It's just that for the child, these two elements have somewhat different meaning.

Need #1: Versatility

Versatility means that I can do with it whatever my mind imagines I can do with it. Sand, mud, water, and sticks are eminently versatile. They are also the favorite toys of any child. There are some basic ways to provide that same versatility in your game.

Provide tools, situations, and environments that have multiple uses. When you design an object, a tool, or an environment, don't just think about what you intend the child to do, think about what the child may *attempt* to do. And then make that possible.

If you provide a hammer for building, make sure that hammer can smash things as well. If you provide water for putting out a fire, let that water create mud when it mixes with dirt. This ensures that objects have integrity and that their relationships are well integrated.

Another advantage to making your software flexible is that it loosens the age restrictions. Different-aged children can use objects for different purposesin different ways. Also, a child can grow with the game. They can come back to it a year later and find fascinating new facets to the game that didn't seem to be there before.

Realize that a child is an extreme functionalist. Everything must have a

use. If the child doesn't find an immediate use, that child will have no qualms about breaking the object so that it fits into some use. Call it egocentric, call it narrow-minded, call it reckless vandalism — the child's functional orientation is something that you can take prime advantage of in your game.

Encourage exploration and banish the fear of failure. Remember, the child's way of learning is by trying everything out. The greatest obstruction we can put in the path of that learning process is to feed children the notion that if they try something interesting, they're going to fail.

Let's say you have an aerial view of a world with continents, islands, and bodies of water. You show the child the starting location and allow him or her to move around elsewhere. What makes sense is to move only a short distance from the starting location. The child may try this at first, but eventually will try flying off to a distance. That doesn't fit into your game, so you just don't allow the child to enter that area. You've just discouraged exploration.

Or let's say you've provided a vacuum cleaner. Those are for vacuuming the floor, right? It just so happens, I created one in a game and let the kids test it. I should have known: They tried vacuuming the alphabet blocks on the shelf with it. Nothing happened, and I felt badly. In the next iteration, I designed the vacuum cleaner to suck the paint off the letters on the blocks. Adults didn't try doing that, but the kids love it.

Rather than restrict the child to linear game flow, reward exploration and novelty. If the child tries to enter an area of the map that is not yet accessible, don't just block entry. Provide some feedback in terms of vital information to the game. If you provide the child with a pen, allow the child to stab holes in the paper as well as write on it. If your background is made of clay, allow the child to smudge parts of it. All these things are vital methods to the child's learning strategy. They also make things infinitely more fun.

Avoid arresting control from the child. This very common failure is found in the best of kids' software. In the early days of procedural code, it was excusable. Today it's not.

We all know how we feel when a process takes over our machine and doesn't allow us to do other work. We're reminded that, "Oh yes, there is a machine here. And right now it's getting in my way." It doesn't feel like fun and it doesn't feel empowered.

When an animation or any process begins — no matter how entertaining it is — the child shouldn't have to wait until the process is over to try out something else. Conversely, trying out something else shouldn' mean aborting whatever is already happening. That's sending a strong message that "You're not really in control here" to the child. Sure, that means a lot better management of your code and a lot more quali-

The Case for Inappropriateness

A major concern I have in leaving my child alone with educational software is a lack of inappropriateness. This becomes of special concern when dealing with the very young (under seven years of age). Inappropriateness is a very vital element at that age. The fact is inappropriateness is the small child's most powerful learning tool. It allows a child to pick up any object and try anything with it. Mud can be cake. A block of wood can be a doll. Underpants can be a hat.

Then, around the age of seven or eight, something very dramatic — and tragic — occurs. It occurs almost universally, in every country and in every culture where such things are observed. Mud becomes mud. Blocks of wood become blocks of wood. Underpants come to have but one use. Anything else is inappropriate.

Show a six year old a rock and ask him what he thinks it is. You could be in for anything. Wait a few years to ask him again, and it's a rock. And only a rock.

It seems something innate to the human species. Only a smattering of individuals manage to escape this syndrome, preserving their sense of inappropriateness into adulthood. I don't know how fortunate it is for those individuals — or for the people that have to live with them — but for humanity the dividends are bountiful.

I doubt we would have mathematics, for example, if it weren't for these recalcitrants. Mathematics is all about inappropriateness. You apply the same set of digits and formulas to entirely diverse sets of realities. And how about original art and social reform and... well, just about anything else requiring nonlinear thought? Whether Newton or Einstein, Beethoven or Picasso, Spinoza or the Lubavitcher Rebbe, Karl Marx or Groucho Marx, it was the child alive within them that made those quantum leaps in human thought.

What we want to do, then, is to parachute the child gently into adulthood, holding on tight to that willingness to try the improbable, the preposterous, and the patently absurd. We must always leave open the option to try out the ridiculous, and even encourage it.

Software just doesn't lend itself to this sort of thing. Programmers don't like users who muck about. We like to design controlled environments, where the user becomes just another fairly predictable object. When you stop to think about it, little kids are a real pain for all of us in this industry.

That's why we write for "good" kids. Kids who will press the right button and only the right button. Kids who like to be rewarded and don't like to get things wrong. Kids who are fairly predictable — just not quite as bright as us big people. But also not as promising as the nudnik trouble makers.

Nobody's going to make a killer app for early learning this way. What we need is not nice software for good kids. We need great software for rotten brats. We need products where kids can discover that by doing completely unexpected and inappropriate things, they can get really nifty things to happen. We need software where the best solution to a problem is the craziest one.

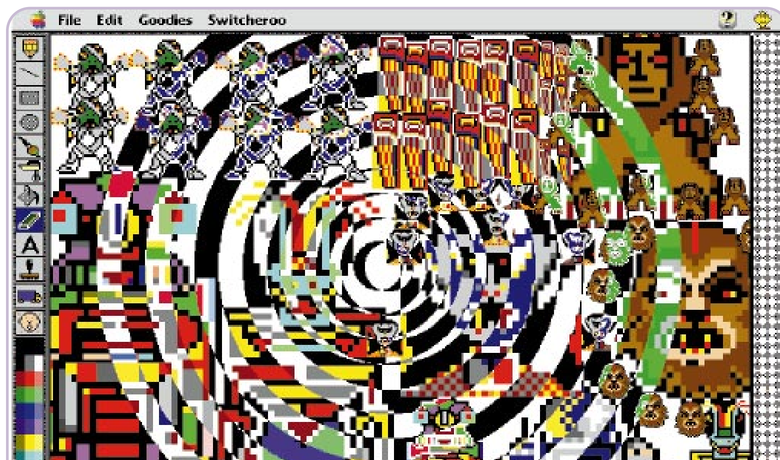
After all, isn't that just what good ol' Albert did when he decided that everything is relative except for the speed of light, that mass and energy are really the same thing, and time is just another dimension? Sounds pretty crazy to me. No wonder he did so lousy in school. He'd do even worse on Reader Rabbit.

ty control, but the playfulness it adds to your game is well worth it.

Knowing that, here are two caveats: Don't make it too easy for the child to abort a process inadvertently. Remember that children will often click the mouse for no apparent reason. But don't throw in some obtrusive interface, either. Come up with something elegant. Also note that there are situations in which allowing a child to abort a process could severely complicate matters and is simply not worth it (this occurs in adult software, too). But it's a situation to be avoided.

Watch out for "The Clickies." This is a typical syndrome suffered by many kids new to computers. In its most extreme form, it involves an almost continuous clicking of the mouse on anything that looks clickable. More commonly, it means that no button survives without at least a double-click — if not a triple or quadruple.

In many cases, this can prevent the child from mastering your game. For example, if the first click of a button initiates an action and the second cancels it, you can imagine how much fun we're having. Similar problems occur when a button leads to a new scene where another button appears in the same place.



Kid Pix (Broderbund)—long-time king of the child software market—allows kids to do the wacky and ridiculous. It was created by Craig Hickman with his 3 year old on his lap. When Broderbund handed it over to an outside contractor for version 2, they wreaked havoc.

Give the kid a break: Test all clickable objects by double-clicking them. You may simply want to clear the event queue of mouseclicks after processing any click.

Don't build in "wrong" responses. This is where so much "edutainment" beats its way to the grave. The last thing any child wants to hear is a machine telling her she's wrong. Up until that point, you had a chance of convincing the child that she controls this mega-power monster.

Let's return to the vacuum cleaner that I made. After the children were empowered to suck the paint off the alphabet blocks, they were concerned about how to get the color back. So I made a holding area where the letters reappeared when vacuumed away. When passing the mouse over these displaced letters, they screamed, "Help! Put me back!" If you dragged them back to their original spot, where they fit in perfectly, they would snap into place. But if not, they would just fall back to the holding area.

This feature provided a great opportunity for kids to learn the shapes of the alphabet characters. Then came the acid test — which turned out to be one of my sweetest moments of success. It was one of the most lucid and revealing windows I've ever had on a child's mind.

I sat a three year old at the machine who didn't know the letter "A" from an ink spill. He vacuumed the letters. He expressed concern. He found the lost letters and dragged them back — without error. Then it happened: He was dragging the capital letter "O" back to its place when he passed over the letter "Q" block. Then he turned back. He lined up the "O" over the "Q" block without releasing the mouse button and said, "Hmmmm. What if…?" And then he let the mouse go. The letter fell back to where it came from. His reaction? "That's funny!" And then he dragged the letter straight to the "O"

Four Subtle Ways to Provide Instructions Without Text

1. Make it obvious

This is the ideal, but most difficult method of providing instruction — the sign of a masterpiece. Create an environment where the child will determine intuitively what is to be done in each situation. Borrowing familiar objects and dynamics from the child's world will help a lot.

2. Characters that talk to themselves.

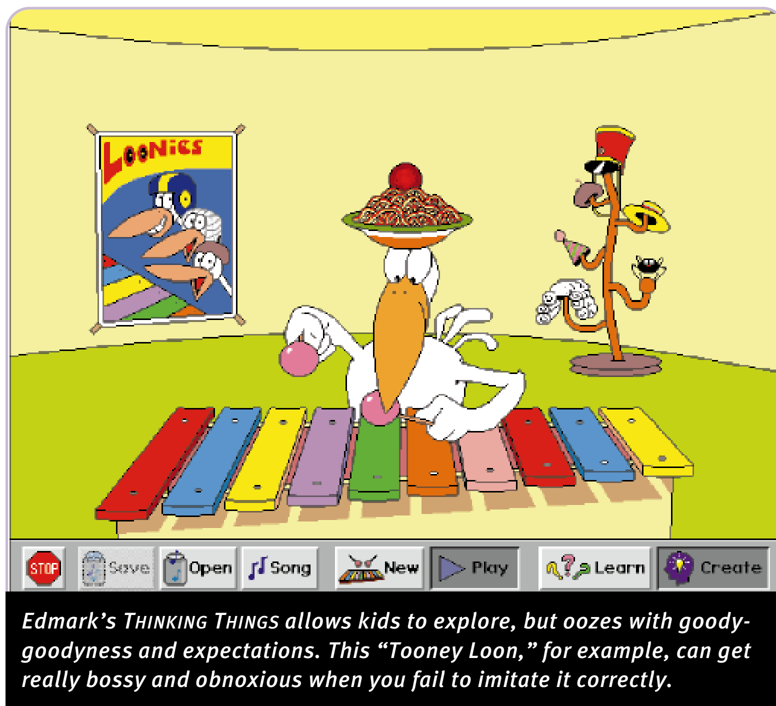
Instead of telling the child, "Now do this," have one of the characters (or objects) talk to itself about what needs to be done.

3. Teach by example

This is a common strategy in 2D scrollers. Have some other character — even the enemy — do what you would like the child to do. Repeat it until the child picks it up. Children like to imitate.

4. Encourage exploration.

Make your environment free and comfortable enough that children will be willing to try things out until hitting on what you want them to do. While they're figuring it out, they'll still be having fun.



Edmark's THINKING THINGS *allows kids to explore, but oozes with goody-goodyness and expectations. This "Tooney Loon," for example, can get really bossy and obnoxious when you fail to imitate it correctly.*

The greatest experience that we can provide for a child is the sense that nothing is as important as the imagination and this game is an opportunity to explore it.

block where he knew it was supposed to go in the first place.

What exactly the child learned from this exercise is a matter of conjecture. But suppose the machine had been inane enough to tell him, "That's wrong, try again!" I know exactly what he would have learned: Experiments are wrong.

DON'T LET THE MACHINE BOSS THE KID AROUND. Let's get this straight: Machines are tools. Like hammers, screwdrivers, automobiles, and telephones. Tools are things people use. A good tool is one that feels transparent in its master's hand. Now, imagine a hammer that looks up at you and smirks, "Hey, look who thinks he knows how to hold a hammer!"

The greatest, and perhaps most seductive error that we can make with children's games is to make the machine into an entity. No cutesy "I don't understand that" or "Now do this" or "That was wrong, try again." The greatest experience that we can provide for a child is the sense that nothing is as important as the imagination and this game is an opportunity to explore it.

Occasionally, there is a need to instruct the child how to use the game. There are plenty of ways to do this without commands (see "Four Subtle Ways to Provide Instructions Without Text").

Need #2: Comfort

Every game designer is familiar with the principle of balancing challenge with ability. What many designers don't realize is that there is a third factor in the formula that allows you to widen the distance between ability and challenge: the player's sense of security. Simply put, if the player feels insecure in any given situation, that player will be less likely to take on challenges. With greater security, confidence increases and more challenges can be faced. More challenges means more time and enjoyment of your game. Comfort and security can sometimes be difficult to provide for little people.

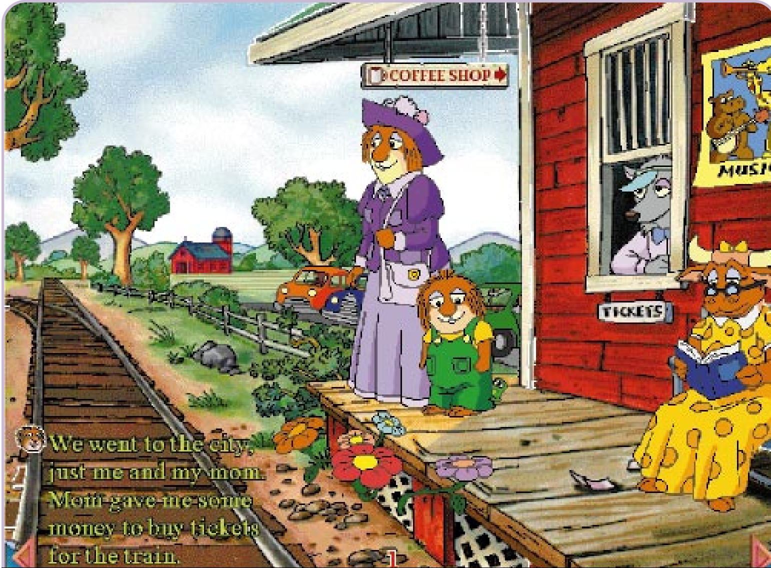
Here's an example: Say you're a little kid. You've watched your parents break down in tears over something that you innocently retooled, redecorated, or otherwise abused. You haven't yet got it figured out, but it's becoming apparent that at a certain point things break, and some (apparently random set) of those breaking incidents cause big people to get really angry, which can imply significantly negative consequences for defenseless innocents such as yourself. Now you view your father's very nifty machine as one of those anger-triggering-when-busted things. You might not feel so secure.

As if that's not enough, when you get into this game, your head starts spinning. Things keep changing. Each screen is different from the screen before. Tools work differently according to when you use them. Characters change their roles and behaviors. It seems that at any moment, the whole thing could just blow up.

Sure, there are kids who aren't intimidated easily. But all children work better when they feel more secure. A game can provide security by following a few simple guidelines.

PROVIDE A GUIDE. Humongous Entertainment does a great job of providing warm, nonthreatening guides with whom every child can identify. Every child who plays PUTT-PUTT JOINS THE PARADE leaves the game believing that he or she is Putt-Putt, which provides a lot of security. There are a number of ways Humongous achieves this. First of all, Putt-Putt always seems to knows what's happening. Putt-Putt never looks too frightened or overly perturbed by events, so neither does the child. Putt-Putt responds to situations just as the child would. Putt-Putt never tells the child what to do or passes judgement on the child's actions. He never behaves in a totally unexpected manner or does anything without the child first approving. He never gets hurt or dangerously lost. If anything negative did happen to Putt-Putt, the child's sense of security and willingness to continue the game could be adversely affected.

(Putt-Putt was Humongous's first — and very successful — attempt at an adventure game for kids. Their second title involved a teddy bear wandering around a house at night in the dark and encountering some exasperating



The Living Book Series (the oldest CD ROM series for kids, originally from Living Books, now at cheaper quality from Big Tuna Productions) provides small children a simple, comfortable environment. It flunks out, however, in versatility. You can't even abort an animation, and some of them are very long.

situations. It was the first child's game I ever saw that could make a child break down in tears.)

A lot of consideration, discussion, and testing goes into creating a guide like Putt-Putt. After all, the guide becomes a central feature of the child's experience. Many developers try to make their guides asexual, androgynous, or totally ambiguous. My sense is that as long as the character is clearly defined early on in the design process, it will integrate well and work. Also, keep in mind (I need my bodyguards present while I make this statement, but it's true) that many boys will have difficulty identifying with a girl character, whereas most girls will have no difficulty identifying with a boy. If it comes down to using one or the other, use a boy.

Another strategy is to provide a choice of guides. This considerably increases your workload, not just in terms of code, but in ensuring that every scene works appropriately with either character as the guide. Beware that it also detracts from the game's focus and feel.

BE CONSISTENT. Just as your guide must be consistent, your artistic style, sound effects, backgrounds, and object behaviors must be, too. Think through everything and define styles and behaviors clearly in your design document.

Children approaching seven years old are just learning to believe that there is consistency out there; that there are laws of nature that you can trust to be the same today as they were yesterday. Half of their play is all about discovering this wonderful, secure consistency of nature. Don't be the one to undermine that.

Many developers try to make their guides asexual, androgynous, or totally ambiguous. My sense is that as long as the character is clearly defined early on in the design process, it will integrate well and work.

One place consistency tends to break down is when the operating system's user interface raises its hoary head. Adults are expected to recognize and deal with Open File dialogs and such, but not children. Make sure to provide your own substitutes to these things if necessary.

MAKE IT INTUITIVE. Once you've learned how to use something, if it looks like it should be used differently you continue feeling insecure about using it. For example, I move my kids about in a beat-up '86 Toyota van. To lock all the doors in this brilliantly designed vehicle, you pull up a switch by the driver's seat. That makes all the locks go down. Brilliant, eh? After all these years, I still get it wrong one time in every five. If that's so with adults, it's all the more so with children.

Of course, what's intuitive to adults may be counter-intuitive to children. To an adult, it makes perfect sense that if you look up, the shampoo won't go in your eyes, or that you place the right arm in the sleeve on the left side of the jacket facing you. To children, these are things to be taken on blind faith out of the awe that they have for these big people who seem to know what they are talking about.

Which means that there's no substitute for testing your interface ideas on real, live children. Show them objects and ask, "What do you think this does?" Then listen carefully as they give their own ideas of what things should look like.

AVOID SHIFTING MODALITIES. Shifting

The Humongous line provides heroes that kids can easily identify with and also serve as guides. The age-appropriate problems are cleverly presented. The user-interface is simple, intuitive and consistent.

modalities means making things work under one set of rules in one instance, but under another set of rules in another. Engineers just love designing things that way. End users are completely confounded by it.

One of the hardest objects I've had to design is a telephone for children. After the typewriter, the telephone is the most poorly designed device of the modern era. It behaves one way when "on the hook," another way when "off the hook," and yet another way while "on line." No wonder teaching young children to operate a telephone can be so frustrating. Ever had a child answer a long-awaited call and hang up while he goes to look for you? Or had a child start dialing the phone while you're still on the line? Every tool that you give the child should have only one way of working, regardless of what happened just beforehand. Every object should have a well-defined, internally consistent set of behaviors that never shifts about.

Usually, when I make a tool for a child, the initial iteration has more than one modality. It takes some thought to find a way to provide everything you want in one simple mode, but you end up with a far easier-to-use tool.

NO READING. Yes, it sounds obvious, but developers keep on doing it. I've even seen software that purports to teach basic reading, but can't be understood unless you know how to read (and even then…).

Know that if a child cannot read, that means the child can't read. You're going to have to find other ways to communicate. Even if the child can read, he or she may not comfortable doing it.

USE KID'S VOICES. The only excuse for using adult voices in software is that it is much easier than using children's voices. Children can be a real pain to record. If they don't get it right the first time, you've usually lost it for the day. Very often you'll have to blend two or three recordings together to get what you want.

But if you do it right, the rewards are phenomenal. There's just no comparison to hearing the fresh, vital, and expressive voices of young children speaking out of your machine. You'll capture the heart not just of the child who's playing, but of the adult who's machine is being hijacked as well.

DON'T INCREASE DIFFICULTY WITHOUT THE CHILD'S APPROVAL. Edmark is notorious for this. Some educator applied half-baked notions of Mastery Learning mixed with poor arcade-game design and arranged gameplay so that as soon as a child gets too comfortable, things become more difficult.

Now, that's often fine in the real world, where there are plenty of signs to tell kids when they're improving and that things are going to get more difficult; generally, an intelligent adult or friend is giving real feedback and reading stress levels loud and clear.

But when progress is blindly automated, it's bad. The child thought he was doing well, and now he's failing. And he can't tell why. The least you can do is explicitly let the child know that difficulty is increasing, as in a typical shooter, where you can see the opponents are becoming bigger and more threatening. You know you're on a higher level.

Personally, I'm a proponent of letting the children decide when to take on bigger and more difficult tasks. This way, they retain their sense of control and develop a sense of their own capacities as well. That may be too much to expect at a very early age, but as they come closer to "metacognition" (awareness of what they are thinking) it makes more sense.

MAKE THE CLICK AREAS BIG ENOUGH. Kids are not as detail oriented as adults; for children, close is good enough. Make sure your buttons are big. If the child is supposed to drag things to specific places, expand the "hot" area invisibly beyond the graphic.

It's also a good idea to provide plenty of feedback when the cursor is over a hot spot. Changing cursors, animating the area, cueing audio, and showing eyeballs that follow the mouse are all effective means of helping children follow the action. These tactics also help them keep track of the cursor's location; losing the cursor is another common phenomena with small children.

PROVIDE IMMEDIATE RESPONSE TO ANY ACTION. Adults begin to feel uneasy if they've clicked a button and nothing's happened within 0.2 seconds. With kids, results are magnified. I previously mentioned the "clickies" syndrome. This is often brought on in otherwise perfectly balanced kids by repeatedly unresponsive buttons. Aggressive behavior has also been noted. They may just plain give up. (Just be thankful you're not programming for primates. In a project creating buttons with icons recognizable by a gorilla, the beast was noted to trash machines that did not respond immediately. Literally.)

Remember that small children have just understood the notion thaty there is such a thing as cause and effect in our world. They're still quite suspicious of the phenomenon and ready to attribute anything to magic. We want them to know that machines aren't magical. Kids are.

There's just no comparison to hearing the fresh, vital, and expressive voices of young children speaking out of your machine.

Build It and They Will Learn

Kids have been getting the short end of the technology stick for too long. Adults — and big kids — are handed more and more power every day, but small folk just get pushed around more and more. Game developers have the tools to empower and liberate children.

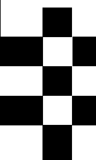
Nevertheless, you have to know them very well. Don't ever assume that you've got children down pat and can predict their every turn and click. Keep studying them closely and they'll never cease to amaze you. ■

Tzvi Freeman teaches Game Design and Documentation at Digipen School of Computer Gaming in Vancouver, British Columbia, Canada. He has designed several commercial games and has acted as a consultant on many others. He can be reached at TzviF@aol.com.

Child Psychology 101 for Game Developers

Many folks think kids are cute. But looks can be deceiving. Here are some of the more extreme examples that indicate just how different those little brains are from ours.

	Adults	Children	Application
Appropriate Usage	Clothes are for wearing. Food is for eating. Toys are for playing with. *Example: you wear socks, eat spaghetti and play with sand.*	Clothes, food and toys are for eating, wearing and abusing. *Example: you can eat spaghetti, eat a sandbox, and eat your dirty socks. Or you can wear the dirty socks, wear the sandbox and wear the spaghetti. Or you can abuse all three. (Adults call this abuse "playing.")*	Your objects and environments must be both flexible and resilient, able to perform all feats they appear able to perform and more. Code must be ready for entirely unpredictable users that could do anything, anytime—including the outrageous and abusive. You must reward it, too
Procedures	Follow the most efficient and logical order of activities. *Example: Go to the toilet. Now pull down your pants.*	Do what you can now and then worry about later. *Example: First pull down your pants, then walk across the house to the toilet.*	Avoid linear procedures. If you have to, then use no more than two steps. Allow steps in any order, even if you find it counter-intuitive.
Context versus Laws of Nature	Things have consistent behaviors and properties, regardless of their context. *Example: Four ounces of water is four ounces of water, no matter where you put it.*	Context is everything. *Example: Four ounces of water in a fat glass becomes more when you pour it into a tall, thin glass.*	Change visual context when the rules change. Be cautious about changing it when the rules stay the same. Don't expect the child to assume things will work the same way everywhere, until it is demonstrated many times.
Process versus Product	Work towards a goal that you value. *Example: You work for hours making something look beautiful, so you value and treasure it.*	It's the journey that counts (until age 5+) *Example: You paint a masterpiece and trash it. Pile up blocks and knock them down. If you are disturbed by someone destroying your work it's not because of its value, but because you made it and they smashed it.*	Make the production process fun. Help the child learn the value of the end product by providing easy ways to preserve and retrieve what they've done. Don't just dump it into the cyberdump.
Abstract versus Concrete	Things can mean something other than what they are. *Example: Adults can usually read and interpret maps and charts.*	Things are what they are which is what they are. *Example: Maps and charts are weird pictures.*	Don't expect kids to read maps or understand charts. Use only very concrete metaphors. User-test all icons, and so on.
Enjoyment Response	Rarely afraid of overload. *Example: Adults who find a game really stimulating and stay with it.*	Unpredictable response to highly stimulating experiences. *Example: Finds a game real stimulating, so the immediately quit to come back later.*	Book lots of time and patience for user-testing. Don't overwhelm the child. Allow them to "take the foot off of the accelerator."
Objective	Wants to be a child again.	Wants to be an adult.	Make them feel big.



SOAPBOX by Nancie S. Martin

Take the Y Out of Computer Games

It's getting pretty bad for some guys out there. First, women wanted the remote. Then there was Title IX and that equal pay thing. But now it's getting serious, because we want your mouse, and we want it now.

In increasing numbers, women and girls are taking over the home computer (we already were the preponderant users of the ones in offices). We're buying and using software like Mattel's BARBIE FASHION DESIGNER, which as of this writing has sold over a million copies in less than a year and is available in 13 languages worldwide. We've taken to e-mail as enthusiastically as we picked up the phone. A man practically has to elbow his wife and daughters aside to get some quality time with his flight simulator these days.

This trend isn't stopping in the family room. Women are designing, producing, and funding software, too. We're hiring some of the industry's best artists and programmers. You'll find us just about everywhere there's a high-end CPU, not to mention landing a rover on Mars or playing pro basketball.

What's your average computing-era caveman to do?

For starters, recognize that XY chromosomes and binary code aren't an invariant combination. Banish the sentences "Women don't like technology" and "Girls are afraid of computers" from your lexicon.

Women and girls like technology and computers just fine. But until recently, leisure-time software hasn't offered us much of interest. And that sad fact has driven our choices.

Men and women don't tend to like the same movies or TV shows, but no one says women don't like entertainment. Boys and girls don't generally enjoy the same toys, but no one declares that girls don't like to play. So why would anyone assume that just because many females don't enjoy DOOM and its ilk, we're not interested in computers or computer games?

It might help to re-examine the whole notion of what a "game" is. In our industry, the term is commonly used to mean a fast-moving, competitive affair with just one winner, be that human or machine. But *The Merriam Webster Dictionary* gives its first definition of the noun "game" as "amusement, diversion" and doesn't get around to "contest" until its fifth definition, just ahead of "animals hunted for sport or food."

Don't get me wrong here. Digital jousting, even if it barely edges out blowing away Bambi, is a fine thing. But there are lots of different kinds of amusement and diversion.

The amusements that girls and women seem to prefer have accomplishment, creativity, and communication as their hallmarks. Last year's titles BARBIE FASHION DESIGNER, BARBIE STORYMAKER, and BARBIE PRINT 'N PLAY address those overlapping needs respectively. Shameless plugs aside, though, if you look at other products that have been successful with women, such as TETRIS and MYST, you'll see those same elements.

You'll also see another common trait: the exploration of context.

People with XY chromosomes have dominated playtime on the PC, but it's not just a man's world anymore

Developmental psychologists see this as a key difference between women and men, and one that informs many of our perceptions and rela-

CONTINUED ON PAGE 53.

Continued from page 54.

tionships. (A forthcoming example is Adventures of Barbie: Ocean Discovery, a scrolling underwater quest that gives the player a mission and places it in a highly explorable environment in which the player's actions have observable effects on the game's characters.)

Developmental psychologists see "context" as a key difference between women and men, and one that informs many of our perceptions and relationships.

These elements all make perfect sense in a nonlinear medium, and in fact, many classic PC and console games have made use of them. So why have most game developers and marketers ignored or only made half-hearted attempts to make products for girls and women — who are, after all, the majority of the population?

The reason is that until recently, most game developers and marketers have been male. Shortsighted from years of staring at monitors and blinded by their own enthusiasms, they've had trouble lifting their heads to see that there's a whole other world out there filled with people who don't look like them and don't think like them. Consumer research, in many cases, seems to have been limited to, "Hey, dude, whaddaya think?"

Our friends in the retail business, also mostly male, didn't do much better. If women weren't showing up to buy software or hardware, clearly they weren't interested. Why bother trying to reach out to them?

To be fair, of course, companies with recognizable brands and licenses with female appeal didn't always recognize their interactive potential. And since developers didn't really know what to do with those brands, there were some odd results, such as the shoot-'em-up cartridge games of a few years back that couldn't have been any more pink or any less interesting to little girls.

But now we have it all: smart purveyors, a computer-literate audience, and retailers who've tasted success (for a while, Barbie Fashion Designer outsold Command and Conquer) and would like more of it. Now all we — and anyone who wants a piece of this market — need to do is to pay attention as girls and women assess all their newfound options, and keep making good stuff.

So hold on to your mice, guys. Or maybe, if you're good, we'll let you borrow ours. ■

Nancie S. Martin is Director of Girls' Software Development at Mattel Inc. in El Segundo, Calif., and the executive producer of many Mattel titles, including Barbie Fashion Designer.