gd

GAME DEVELOPER MAGAZINE

## DECEMBER/JANUARY 1995


UBM Tech

GAME DEVELOPER

# The Dark of the Electric Pickle

In the end, I didn't even have a chance to light my pickle. In my best shot yet at Andy Warhol's promised 15 minutes of fame, I blew it, but it all might have been different had they just given me the chance to demonstrate to the world the miracle of the electric pickle.

I'm often the target of joke e-mails purporting to be from various people, but I couldn't dismiss the one purportedly from *Danny!*, the daytime talk show. *Danny!* (exclamation point mandatory), of course, is a vehicle for Danny Bonaduce, who, if you're a certain age, you'll remember as the red-haired and mysteriously edgy moppet from *The Partridge Family*.

## Danny Who?

Yes, *The Partridge Family*, a show that has been strangely unwelcome in the current "rediscovery" of the 70s. Unlike the *Brady Bunch*, there's been no *Partridge Family* movie, no stage productions, not even a regular slot on Nick at Night. Why? Because the Bradys played to the status quo, then and now—sprawling suburban house, servants, an affluent and secure whitebread home in which sons and daughters happily embrace the bourgeois.

The Partridges, on the other hand, were darker. In the cluttered, claustrophobic garage in which they practiced, they gave birth to songs, not about "Sunshine-y Days," but about doubts and fears. "Stop! I Think I Love You"? Why couldn't he be sure? What was David Cassidy saying? That corporate America, by co-opting the language of romance, had stolen the emotional compass to the point where none of us, in fact, could go forward with confidence. The Bradys on the other hand, told us that, "When it's time to change, you have to rearrange who you are and what you want to be." Sha-la-la, indeed.

The Bradys had station wagons and convertibles, symbols of consumption and status. In contrast, the Partridges had that most poignant of all symbols of freedom from the status quo—a schoolbus with birds painted on the side.

Where the Bradys had an emotionally stunted servant woman in Alice (why did she need Sam the Oh-So-Blue-Collar Butcher to validate her worth?), the Partridges portrayed a much more complicated world, in which one has to dance with the capitalist devil even as one decries its excesses. This complex enigma was embodied in the work of one actor—the young Danny Bonaduce.

So, of course, I returned the call.

## A Genius *and* A SCUBA Master?

There was to be a show on brains vs. brawn, and my pro-nerd work over the years had not gone unnoticed. "What makes you think you're so smart?" they asked me. "I don't think I'm smart," I answered, "I just edit a magazine for smart people." "It'll have to do," decided Producers Julie Knapp and Rita Whack. "What can you do for the talent competition?"

My immediate answer: "Program a computer faster than anyone who's better, and better than anyone who's faster," drew a stony silence. I tried sports: "Throw a Frisbee-brand flying disc 120 yards fairly accurately?" Not in a studio. "Hold my breath while wearing diving gear?" Not exactly riveting television. "Ride a mountain bike over a moderately sized stick?" Stony silence. "I can juggle a little. Just three balls, though." There was a long sigh and I could hear the sound of a hand rubbing

a forehead. "Bring your SCUBA gear. We'll figure it out."

Breaking deadlines left and right (sorely testing the good humor of managing editor Nicole Claro and, over at *Software Development*, Barbara Hanscome), my wife Tina and I flew out to Chicago Thursday night, the night before the taping. Impressive as it was to be picked up at the airport by a limousine, it was less impressive that the driver had lost the car in the parking structure, missed the exit from the airport, and got lost again on the way to the hotel. I mean, I don't know Chicago, but does it normally take two hours to get from O'Hare to Evanston with no traffic?

Three-and-a-half hours of sleep later, it was time to get up and get to the studio. Finally, I would know if green rooms were really green and if television sets had as much good food as movie sets. Tina and I debated whether I should drink two or three cappucinos to give me a little bounce or whether the resulting quaver in my voice would come across bad on television. I didn't have the choice. By the time I got to the (not) green room, I couldn't even find real cream for the dregs of a pot of Maxwell House coffee.

## Show Us Your Pickle

I was introduced to my fellow "brain" team members, and it was here that things started falling apart fast. After meeting Alan, the systems analyst, and Charles, the biochemistry graduate student, I was introduced to Quentin and Gary, two stand-up comedians who were planted in our panel apparently because the producers weren't sure we'd be amusing enough. Quentin would eventually "win" as the most interesting of our team, despite the fact that his "talent" was telling an offensive joke that was stupid when Eddie Murphy first told it 10 years ago. With his chunk-gold jewelry and mock inner-city speaking cadences, this guy was popped straight from the Play-Doh Create-A-Comic kit. When he finally got on TV, the first words out of his mouth were, I swear to God, "Hey, how y'all doin'? Some fine-looking women in the audience here!" Then, to put the audience in stitches, he acted effeminate and waggled his tongue suggestively. Quentin had appeared on several talk shows, which is apparently what young comics do nowadays rather than actually develop a witty routine.

> The Partridges were darker than the Bradys and sang songs about doubts and fear. "Stop! I Think I Love You"? Why wasn't he sure? What was David Cassidy saying?

Still, the stand-up comics at least knew the talk-show drill, which gave them a distinct advantage over me. I'd brought some skin diving gear but had no idea what to do in it. I gave some thought to duct-taping my mouth and nose shut and doing push-ups to show that one could be fit without being bulky, but I didn't even know if the production staff could find duct tape in the few remaining minutes before taping time. It was then that I was saved by the electric pickle.

One of the guys replaced by the idiot comics was willing to lend me his pickle and a lamp cord. As I'm sure you know, when 110 volts of alternating current flows through a pickle, the result is a coronal discharge that dances like green neon, what the Romans would have called an *aurora cucumeralis* if only they'd spent a little less time building aqueducts and a little more time installing cheap, universal electrical service.

So I reported to the backstage area newly charged with confidence. There are few things that one can know with metaphysical certainty, but one is that electrifying a pickle on national television would be a "grabber." It's got all the elements of great drama: surprise, danger (electrocution and explosion being distant, but real, possibilities), and visual appeal that can't be beat.

Then, sad to say, it all came apart. Intimidated by the jeers of the crowd, tongue-tied by the insistence of the staff that I "know what you're going to say before you go out," tired and cranky from lack of sleep, I went on and, in a word, bombed. I was asked to step aside, to support my team, to be seen, perhaps, but not heard. And then it was over. The judges decided that Jose, one of the "Macho Maniacs," was the most interesting of us all, a decision I couldn't disagree with, based on what made it onto television.

But I can't help but think what might have been. I can't help but dream of the electric pickle. ■

**Larry O'Brien**
**Editor**

*When Larry O'Brien is not singing* Brady Bunch *and* Partridge Family *tunes, he can be found at* Game Developer *magazine or attempting to electrically charge various vegetables.*

# Gaming for The Fun of It??

**SAY IT!**

The shady staff of *Game Developer* would love to hear your comments, questions, and suggestions! Please send them to: *Game Developer* magazine, Sez U!, 600 Harrison St., San Francisco, Calif., 94107. For those of you who *do* have access to the Internet, send e-mail to 71743.452@compuserve.com or go to the *Game Developer* web site at http://www.mfi.com/gdmag. Thanks!

**Dear Editor:**

I just have to say I really enjoy your magazine. I've made a living doing defense and commercial product design for many years and have only now gathered the resources to do what the majority of your readers are already doing—creating games! I've tested the waters by doing game-testing, consulting, manual design, editing and much playing. Now I'm ready to see if my hare-brained ideas will sell!

I enjoyed Barbara Hanscome's "Gamin' for Grrrrls" (Chopping Block, Oct./Nov. 1995). Despite the fact that boys and girls *do* have differing tastes, we are after all, both *human* and have that much in common. We like to be entertained, challenged, amused, thrilled, and teased. Boys don't necessarily need blood and violence, and girls can live without cute, fuzzy animals. My sons not only like the background sound in games, they thrive on it! In addition, my 8-year-old plays all four of the characters in Street Fighter 2, including the girl (she has "neat powers")!

I believe we can most easily find common ground by examining games of the past, in addition to perennial favorites. I have noticed that both sexes really get into games that let them insert their own personalities. Role-playing games let players of either sex use their unique strengths. Hey, men like a good game of chess, yahtzee, cards, and so on–no real violence there. My point is that I honestly think these big game companies are trying *too* hard and maybe they should consult with some *adults* who have enjoyed gaming for the fun of it.

**Randall G. Arnold**
**Coppell, Texas**

## WAIT A MINUTE, MR. POSTMAN

**Dear Editor:**

Please stop calling the postal service's universal access, low technology, information transfer media snailmail! Remember, only 10% of the U.S. population is on the Internet at this time. The rest of them still need to move physical objects around at the universal cost of 32 cents per ounce.

**Jason Feinman**
**Via e-mail**

**Dear Editor:**

How come you don't put articles online, thereby saving trees and many post officers' backs? I know money is a concern, but can't you just stick the advertisers' ad in the articles? I promise to be subliminally affected by their ads.

**Andrew Shebanow**
**Via e-mail**

*Editor Larry O'Brien responds:*
*After receiving the preceding disturbingly angry letter from a postal carrier in response to our referring to ground mail as "snailmail," we're not about to become even more of a target by throwing hundreds, maybe even thousands, of carriers out of work.*

## GIMME MORE!

**Dear Editor:**

I'm a big fan of your magazine, and I'm glad to see you've lowered the price. I thought the October/November 1995 issue was your best yet. Mike Michaels's "Organizing User Input, Part I: The Input Queue Manager and Keyboard Events" was very well written and informative. I do have a suggestion, though: I'd really like to see more information about game industry sales figures and other market data. Unlike other PC application categories, game sales information is very hard to come by. I'd like to know how well games sell, what the total market size is, what typical budgets for games are, and how game developers make their money. After all, being a game developer isn't simply a matter of hacking out code–you've got to sell those games too. Thanks.

**Andy Shebanow**
**Via e-mail**

## LITTLE NIKKI

**Dear Editor:**

I was walking down the street the other day just minding my own business when this woman ran into me and knocked me into moving traffic. Now, I was quite upset until she said she worked on this game maker magazine or something, and gee didn't that sound interesting, and, well I guess I could forget the whole incident if she would send me a copy of what sounded like a great read, but I never heard from her again. I thought she said her name was Nakita Freeman or something, so I looked up game stuff on my groovy web browser and there's someone on your staff named Nicole. I thought she might be working there under an assumed name or something. I don't care so much about the magazine anymore, but I just wanted to warn you that there's a woman on your staff who doesn't look where she's going on the busy sidewalks of San Francisco.

**Enrique Hombrelibre**
**Via e-mail**

*Contributing editor Alex Dunne responds: Thanks for the note. We've taken care of Nicole Freeman, a.k.a. "little Nikita." She travels under a variety of assumed names, especially when involved in covert pedestrian-bumping operations. Rest assured that everyone on our staff was alerted to her slip-up and that she will be severely reprimanded! It's slip-ups like this that can devastate a little magazine like ours.*

## ERRATA

Quality *is* job one here at *Game Developer* magazine. We strive to correct any mistakes we've made to provide you with best product we can. That said, we lower our heads and admit to the following two mistakes.

- In Mike Michaels's Oct./Nov. 1995 feature article "Organizing User Input, Part I: The Input Queue Manager and Keyboard Events," we left out Listing 3, HEAPMGR.C.
- In "Getting Started with VESA" by Matt Pritchard (June/July 1995) part of the code in Listing 2, VESADEMO.C, was omitted.

You can access each of these listings in its entirety (as well as all referenced code from previous issues) on the fabulous carousel that is the *Game Developer* ftp site.

## Our Readers

This month, our readers have much to say about gender differences, the postal service, and suspicious encounters with the staff of Game Developer magazine.

# A Hardware Spec for Games

Alex Dunne

The standards specification under discussion at the Game PC Consortium is a hot topic these days. So hot, in fact, that we've got the results of the vote in this month's Bit Blasts (we are so on top of things).

Last issue, I examined the GamePC Consortium's (GPCC) efforts to put together a benchmark for three-dimensional graphics performance. The GPCC is a young organization comprising hardware and software vendors in the PC game industry. The consortium was formed to target the vacuum in PC game standards through the creation of the GamePC specification—an all-encompassing GamePC standard similar to the Multimedia PC (MPC) specification. This standard is a tough nut to crack, not merely because of the technical hurdles that have to be cleared, but because of inter- and intracompany politics that can hold back rapid adoption of a standard. The three-dimensional graphics benchmark I described last issue is merely one component of this proposed specification.

## Standardizing Game Hardware

Like the MPC certification, which is controlled by the Software Publishers Association's Interactive Multimedia section, the GamePC certification will provide a way for consumers to determine the minimum hardware they need to enjoy their software purchases. It will also help the game development community develop and market


The GamePC Consortium's specification for gaming hardware will be implemented as a seal of approval, indicating the type of hardware recommended for game play. These logos are still in the design stages.

games because a common level of game performance will be expected from the hardware.

Why not simply use the MPC standard for games? It doesn't go far enough. For instance, a graphically undemanding and processor-friendly game of the Myst variety would do fine using the MPC3 specification. A game that required a bit more horsepower under the hood—perhaps Dark Forces or Wing Commander 3—would need a specification that provided for a faster processor, graphics, and perhaps better audio as well. Further down the road, as games demand more sophisticated hardware, the specification can be updated or higher levels of the specification can be created.

## The MPC Spec

To see how the GamePC specification will be used, look no further than the MPC certification as a model. The MPC specifications (there are three levels—the higher the number the more current the specification) dictate the minimum processor, RAM, hard drive, floppy drive, CD-ROM drive, audio, graphics performance, video playback, user input, I/O, and system software present on a PC. In addition, the MPC Working Group provides a test suite on CD-ROM, written by the National Software Testing Laboratories, for establishing whether a computer is delivering MPC3 compliance in the key areas of processing speed, video playback, graphics performance and audio. Hardware vendors are required to pass the test suite in order to display the MPC mark on their products. In 1994, the MPC1 MPC2 certification marks were extended to individual CD-ROM drives and sound cards, and in 1995, the MPC3 mark was extended to video playback boards and speakers.

Because the MPC standard was developed years ago, it has already staked out much of the territory that the GamePC Consortium roams. Creating another specification that dictates similar—but not identical—requirements as the MPC specification will only confuse consumers. To avoid duplicity or conflicts between the certifications, the GPCC has decided to base the GamePC specification on either the MPC2 or MPC3 specification and build on those requirements in areas specific to game play. (You can find a thorough description of the various MPC requirements on the World Wide Web, at http://www.spa.org/-mpc/default.htm.)

As a result, the GamePC specification will rely to a large degree upon one of these two MPC levels as a baseline. In theory it's an excellent plan—consumers won't have to worry about conflicting requirements in the two specifications. With a little collaboration, the two standards organizations should be able to effectively update their specifications on a regular basis without stepping on each others' toes. However, there are some parts of the GamePC ballot that risk conflict with the MPC specification.

## Turning Suggestions into a Ballot

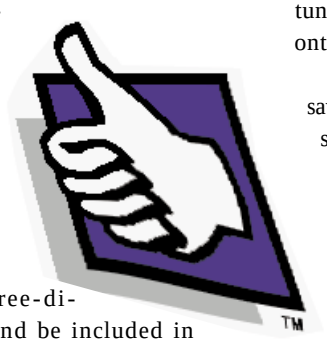
The following items present in the GamePC specification were bandied about among GPCC members. Some items made it onto the ballot for members to vote for or against the specification, while others were shelved for future versions of the specification. Some of the suggested items were pretty cutting edge, but given that the logo won't take effect until sometime well into 1996, the recommendations were appropriate. Here's a quick rundown of items that were suggested for GamePC-compatibility:

*Three-Dimensional Graphics.* The three-dimensional graphics benchmark that I profiled in my last column was probably the part of the specification that generated the most debate. In fact, it wound up being such a complicated topic that it didn't make it into the official GamePC ballot. The group is still working on it as we go to press, and it will be voted on independently from the rest of the hardware requirements. When completed, it will likely take shape as a test suite that computers will have to pass, similar to the MPC test suite.

*Three-dimensional User Input.* A requirement for three-dimensional input (such as a joystick-generated command) was recommended for game machines, as the MPC specification calls only for a standard 101 keyboard and a mouse. The keyboard is fine for Doom and the mouse is great for Myst, but a Descent-type game or a flight simulator allowing six degrees of freedom should require a flight stick to keep players from auguring in. However, the ballot sent out by the GamePC Consortium only queried for the presence of a universal serial bus (USB) digital joystick interface and a front-mounted joystick port—there was no mention of the computer actually having a joystick hooked up. Specifying that a person have a joystick port, but not requiring the joystick itself is somewhat like requiring that the computer have a port for a keyboard, but not requiring the presence of a keyboard (the big difference being that every computer comes with a keyboard, and no computer that I know of automatically comes with a joystick).

A consumer without a joystick is apt to look at a flight simulator box, see the GamePC logo, figure he or she meets the specification ("Hey—I've got the joystick port"), bring the game home, and find out that the darned thing doesn't work too well with a

mouse and keyboard. I may be nitpicking, but I feel that if the specification is going to ask for something as nifty as a front-mounted USB, you might want to require the control that plugs into it.



*Three-dimensional Sight and Sound.* Various GPCC members proposed that information pertaining to three-dimensional sound be included in the specification, and others favor support for stereoscopic devices in the GamePC mark. My feeling when I first saw those suggestions was, yes, those high-end goodies should be in future versions of a specification, but coming off the blocks with a specification that calls for hardware that a miniscule portion of the gamers own (and not many games will support anyway) will discourage consumers from taking the specification seriously. This is one of those cases where the special interests of certain companies can cause political problems in specification creation. Fortunately, neither of those items made it onto the ballot.

*CD-ROM.* This was an item that I saw in potential conflict with the MPC specification. The GPCC proposed a data transfer rate from a CD-ROM at 450KB per second—in other words, a triple speed drive. This raises two questions: why standardize on this rare model of drive (why not just base it on a 4X speed?), and won't this conflict with the MPC3 specification if that gets ratified as the basis for the GamePC specification? To attain the MPC3 rating, a computer must have a quad-speed CD-ROM drive, so approval of a triple-speed drive will cause the GamePC rating to dip below an already approved base standard. Some kind of amendment to the standard might have to be made to correct this apparent conflict.

*Audio.* The GamePC specification seems to be fairly similar to the MPC specification. Game PCs will likely be required to support the current standard sampling rates between 8 and 44.1KHZ and support wavetable synthesis for MIDI playback.

*Communication.* With the head-to-head capabilities of many games on the market these days, the GPCC suggested requiring a 14.4KB modem in a Game-PC-compatible machine. Excellent—although a 28.8KB modem might have been better (oh well). I wonder if specifying access to one of the major commercial online services shouldn't be a requirement of a future GamePC specification. By all accounts, CompuServe, America Online, Prodigy, and the Microsoft Network are hot and heavy to promote a new generation of graphical games using their online services as a backbone. Another idea for future GamePC specifications is a network card— nothing beats the speed of head-to-head play when you're connected to your roommate's computer with some coax!

*System Software.* No mention of system software is made because all MPC specifications specify DOS and 16-bit Windows. However, the GamePC ballot queries support for AutoPlay (or similar functionality), which implies support for Windows 95. The question of which operating system to standardize on could confuse the specification if it is not cleared up. Additionally, GPCC members such as IBM have made it clear that they'd like the GPCC to come up with a specification that's operating system neutral. This means that either the AutoPlay requirement will have to be ditched, IBM will have to enable its own version of AutoPlay in OS/2, or IBM is out of luck.

The results of the balloting will be available by the time you read this. However, it's still not clear when the GamePC specification will begin to appear on game boxes or hardware. Realistically, I wouldn't expect it until the spring or summer of 1996. Yes, there are issues to be resolved regarding the GamePC specification, but there's no question that it's a step forward for everyone—developers and consumers alike. If you'd like to keep up to date with what's going on with the specification, or you're interested in becoming a member of the GPCC, check out their web site at http://www.mmwire.com/gamepc/gpchome.html. ■

*Alex Dunne is contributing editor to* Game Developer *magazine.*

# Let's Hear It for the Boy

Nicole Claro

Christmas is here, and there are new tools, new books, new virtual headsets for everyone! We've also got the scoop on the results of the GamePC Consortium's vote on specifications!

O.J.'s over, the World Series actually happened this year (with the Cleveland Indians, no less!), and my editor got to rub shoulders with our favorite bottom-tattooed, former child star. Judgment Day happens in less than a week, but after that, things could start to get dull around here. Hey, Christmas, a big season for the game industry, is just around the corner! Many of our readers have probably recently shipped projects they'd worked on diligently for months and are ready to spend some time exploring each others work (not to mention the tools they each used on that work). Enter hardware for the Christmas season—Virtual Boy from Nintendo.

Nintendo recently launched Virtual Boy to the tune of a $25-million marketing campaign. The VR headset is a RISC-based, 32-bit system using two high-resolution, mirror-scanning LED displays, which immerses the player in a fantastic world. The player controls the action inside the lightweight headset (fitted with stereo sound) using the double-grip controller with six buttons and two plus-keys. Nintendo was responsible for the three-dimensional image immersion technology used in Virtual Boy, while the company has licensed the proprietary display technology from Reflection Technology Inc., based in Waltham, Mass. Nintendo is currently working with several companies on third-party games designed specifically for the Virtual Boy headset, which will retail for $179.95. Gameboy, Virtual Boy, where does the boy go from here?

■ For more information contact:
Nintendo of America
4820 150th Avenue N.E.
Redmond, Wash. 98052
Tel: (206) 882-2040
Fax: (206) 882-3585

## Diamond on the Edge

Virtual Fighter, "Ready, Go!" Windows 95-integrated video and audio has reached new heights. Diamond Multimedia Systems Inc. has announced a line of integrated three-dimensional multimedia accelerators that lets users play a variety of high-powered games under Windows 95.

The company believes its Diamond Edge 3D is the only single-board accelerator that features a digital gameport for precise joystick control and two video gameports that let you play specialized, multiplayer titles. Several game companies are developing products specifically to take advantage of the capabilities offered by Diamond's new product. Papyrus's Nascar Racing and Interplay's Descent: Destination Saturn are two games slotted to be bundled with the upcoming release of Diamond Edge 3D. Retail price for Diamond Edge 3D will range from $249 to $299.

■ For more information contact:
Diamond Multimedia
Systems Inc.
2880 Junction Ave.
San Jose, Calif. 95134-1922
Tel: (408) 325-7000
Fax: (408) 325-7070

## Crash, Bang, Boom!

Positron Publishing has released the Dynamic Motion Module, the first

physics-based collision detection program for the PC. Designed as a plug-in for Autodesk 3D Studio, the Dynamic Motion Module uses physics to create series of keyframes more precisely than you could by hand. It lets animators combine objects that have been assigned dynamic motion with other objects that rely upon key frame motion and detects the collision of objects (with a collision detection resolution of up to 1/480th of a second), calculating the resulting motion response of the objects. The module uses quaternions (complex algebraic structures) to improve rotation and lets you apply factors such as gravity, wind, acceleration, velocity, and drag to your objects. Fully compatible with 3D Studio's internal splines, the Dynamic Motion Module also provides smooth-shaded preview images as frames are generated.

■ For more information contact:
Positron Publishing
1915 N. 121st St.
Ste. D
Omaha, Neb. 68154
Tel: (402) 493-6280
Fax: (402) 493-6254

## VB Underground

The Waite Group Press has released another in its series of "Black Arts" books for programmers. *Black Art of Visual Basic Game Programming*, by Mark Pruett, is a guide to every aspect of building Windows games from the ground up. Step-by-step tutorials clearly explain essentials like drawing the boundaries of the game playing field, using the Windows API to its fullest to create sprites and bitmap masks, and

## HOT OFF THE PRESS!

**It's the moment we've all been waiting for...Just days ago (remember, it's December in Magazineland, but October everywhere else) the GamePC Consortium (GPCC) announced the results of its vote on standards for game specifications. For background information on the debate, see Alex Dunne's Crossfire column, on page 9 of this issue. Following are the standards the GPCC agreed on for the GamePC Level 1 Compatible System Specification. The specification covers five areas, systems, CD-ROMs, graphics, sound, and video. The GPCC also agreed on a Recommended System Specification, but due to its length, we couldn't provide that data here. Go to our soon-to-be web site (http://www.mfi.com/gdmag), though, and you can view it.**

**SYSTEM**
- **Must Meet MPC level 2 minimum specifications**
- **Base system must contain at least 8MB RAM**
- **Must Support AutoPlay (or similar functionality)**
- **Must be able to read 10MB data file from hard disk in 17 seconds (more than 600Kb per second).**

**CD-ROM**
- **Must be able to read 10MB data file from CD in 30 seconds (300K/sec).**

**GRAPHICS:**
- **Must run "Fox and Bear" benchmark (640x480x8) at a rate of 30 fps**
- **Must implement local bus (VLB or PCI) graphics with total video RAM greater than or equal to 2MB**
- **Minimum total RAM (base system RAM+ video RAM) must be 9MB**
- **Display and monitor must support the following display modes:**

| | | |
|---|---|---|
| 320 x 200 | 256 Colors | (Mode 13 and Mode X) |
| 640 x 400 | 256 Colors | (Mode 100h) |
| 640 x 480 | 256 Colors | (Mode 101h) |
| 640 x 480 | 32K Colors (5:5:5) | (Mode 110h) |
| 640 x 480 | 64K Colors (5:6:5) | (Mode 111h) |
| 640 x 480 | 16.8M Colors (8:8:8) | (Mode 112h) |
| 800 x 600 | 256 Colors | (Mode 103h) |

**SOUND:**
- **Must be able to play 4, 22KHz 16-bit wave buffers mixed, concurrently**
- **Must support all sample rates between 8KHz and 44.1KHz, mono and stereo**
- **Must support General MIDI compatible wave table synthesis.**

**VIDEO**
- **Must be able to play a sample MPEG movie file from CD-ROM (accurate sound synchronization with no audio breaks)**
- **352 x 240 window in display mode 640 x 480 x 8 must run at a rate of 15 fps**
- **Must be OM-1 MPEG compliant.**

**Systems that the GamePC Consortium tests and verifies to conform to the above specifications can earn the *GamePC Level 1 Compatible* certification mark.**

## Schmoozenews

Stick it to Them...Microsoft began shipping the Game Software Development Kit to developers at the beginning of October. Some of us got betas at the Computer Game Developers' Conference last April. The Gossip Lady has looked in vain in her mailbox but perhaps a lucky reader will tell us what video cards are supported. Since Microsoft's standard answer to anyone with problems running games under Win95 is, "Talk to your [video, sound, joystick, mouse] vendor" this is a crucial question. Oddly enough, Win95 has problems with some Logitech mice. Does Microsoft's entry into the joystick market mean that other joystick manufacturers are going to experience these unfortunate bugs also?

A Learning Experience...Broderbund, always strong in edutainment, is going after that segment of the market in a big way. Not only did it purchase The Learning Company but Broderbund has also started a new line of curriculum-based games under the Active Mind label.

Sony-Sega Title Bout...Someone at Sony needs to give those PR people a raise! Sega, probably after seeing the tremendous amount of ink that the Sony Playstation is getting, lowered the price on its Saturn in an attempt to stay competitive. Talk is that Sony's only possible weak spot in its attempt to wrest the set-top market away from Nintendo and Sega is its lack of game design experience. With the hiring of Kelly Flock to head up the newly renamed Sony Computer Entertainment, this last is no longer a problem. Flock has the guts and the smarts to say what all of us developers have been muttering ever since Hollywood mistook the "play" in "game play" as something that you go watch—not something you do. Flock says, "Anyone who talks about interactive movies doesn't know the first thing about either." Meanwhile, Sega announced it will begin shipping titles for the PC.

Publishers Repel Pirates...Wonder why your royalty checks are low? Maybe it has something to do with all the pirate CD-ROMs. The Software Publishers Assoc. (SPA) just won injunctions against E.V. International, M&S Assoc., Mr. CD-ROM, Softshoppe and Stylin Multimedia, mostly for game pirating. On another front, shareware authors have involved the FBI and the Royal Canadian Mounted Police in shutting down pirate shareware CD publishers. Rumor is that the shareware authors may join up with SPA.

*Talk to The Gossip Lady at 71501.3553@compuserve.com.*

monitoring object collisions. Every facet of game creation is covered here, including graphics, animation, soundtracks, and increasingly difficult game levels.

The 600-page book comes with a CD-ROM with sample games that demonstrate smooth-scrolling backgrounds, using bitmap tiles to create a maze, and adding ray-traced graphics to your gamescape. The CD-ROM also includes 16- and 32-bit reusable Visual Basic code to plug into your own games and all the examples, graphics, and sound effects covered in the book. *Black Art of Visual Basic Game Programming: The Newest and Easiest Way to Create Games with Visual Basic* retails for $34.95.

■ For more information contact:
Waite Group Press
200 Tamal Plaza
Corte Madera, Calif. 94925
Tel: (415) 924-2575
Fax: (415) 924-2576

## Intel for Everyone

Intel has introduced Indeo Video Interactive, a new wavelet-based codec that enables real-time interaction and control of video and graphics imagery in multimedia and game application for the PC. It ups the ante on image quality, offereing far better quality at lower rates than possible before with Indeo Video.

Indeo Video Interactive includes features such as transparency support for interactive digital effect, which lets you overlay arbitrarily shaped video or graphics onto other video and graphics, for run-time interactive control; local window decode, with which you can create an indepentent video playback window within a larger video playback display or graphics scene; random keyframe access; and a scalable quality feature that scales quality between several different quality levels, depending on the PC's CPU capability. Indeo Video Interactive for Windows 95 and Windows 3.1 will initially be available within an Intel SDK, with drivers for Apple's QuickTime 2.01 and 2.1 to come in early 1996 (just about the time this hits the stands).

■ For more information contact:
Intel Corp.
2200 Mission College Blvd.
Santa Clara, Calif. 95954-1537
Tel: (503) 264-6277

*Nicole Claro is managing editor of* Game Developer *magazine. Contact her at nclaro@mfi.com.*

## UPGRADE YOURS!

- RAD Software has released version 2.0 of Smacker, which has over 100 new features. Smacker 2.0 has utilities up to 400% faster than previous versions and is available on CD-ROM. The upgrade is $95 for current Smacker users.
- Autodesk is at it again (they put those upgrades out too fast for us to keep track!) with the recent release of Animator Studio 1.1. This Windows 95-compatible version of the two-dimensional paint and animation software features many new plug-ins and a digital clip library on CD-ROM. Current Animator Pro users can migrate to the newest version for $149; the migration will cost $199 for Animator and Multimedia Explorer users. For new users, Animator Studio 1.1 will retail at around $295.
- Kai's Power Tools 3 (KPT3) is the latest version of the imaging software available from MetaTools (formerly HSC Software). KPT3 is optimized for Power Macintosh and Windows NT and Windows 95 and features several new tools and precise controls over imaging special effects. KPT3 will retail for around $199, and registered users of KPT2.1 can upgrade for $79.

# Texture Mapping Part IV: Approximations

Chris Hecker

So you thought we were done with perspective texture mapping. Didn't you feel something was missing? The penultimate article in this series helps shed some light on the ins and outs of approximations.

Knowing, understanding, and following through on your goals are key parts of software development. Often, I'll go into a project with a perfectly valid set of goals, only to get distracted along the way and produce something that meets an entirely different set of goals, but completely misses my original ones. This phenomenon seems to be pretty common in the game industry as well, where companies go into a project with the goal of creating a great game, but end up creating a nice piece of technology with no game play.

Similarly, our goal during this series is *not* to produce a perspective texture mapper. Surprise! Our goal is actually to draw perspective-texture-mapped triangles on the screen quickly. A subtle but important difference exists between these two goals. As with most things in real-time PC graphics, the result on the screen is the only thing that matters, not how you got it there. We can exploit the difference between what *looks* right and what *is* right and get big speedups in our code.

In other words, if a beautifully written, mathematically perfect texture mapper and a total hacked piece of junk produce the exact same results on the screen (including avoiding jitter and all the other things we've been learning), and the hack is 10 times faster, then the choice is clear if you're interested in speed. It's important to note that this does not mean we've been wasting our time learning about "correct" perspective texture mapping. In fact, it's just the opposite. Now that we intimately understand how the math works, we're in a much better position to throw it all out and cut corners.

## In Our Last Episode...

Let's quickly summarize and tie up loose ends from my last column in this series, "Perspective Texture Mapping, Part III: Endpoints and Mapping" (Behind the Screen, Aug./Sept. 1995). The summary is pretty short: we've developed a complete, high-quality sub-pixel-accurate perspective texture mapper.

The only loose end we've got left (besides performance, which is the main subject of this article) concerns the real-to-integer texture coordinate mapping. When we left off, we had a bug in this mapping and we needed to choose a rounding rule to get the correct mapping. I hinted that we already had the information available to make the decision on which rounding rule to use, but I didn't give the answer. As many of you probably guessed, the gradients are the key to making this decision (which implies you must switch between rounding rules at runtime—and this is indeed the case). Unfortunately, limited space keeps me from going into the derivation of the solution. If you're interested, you can pick up the sample code I mention at the end of this column on the *Game Developer* ftp site. You'll find a big comment block explaining things there.

## Divided We Fall

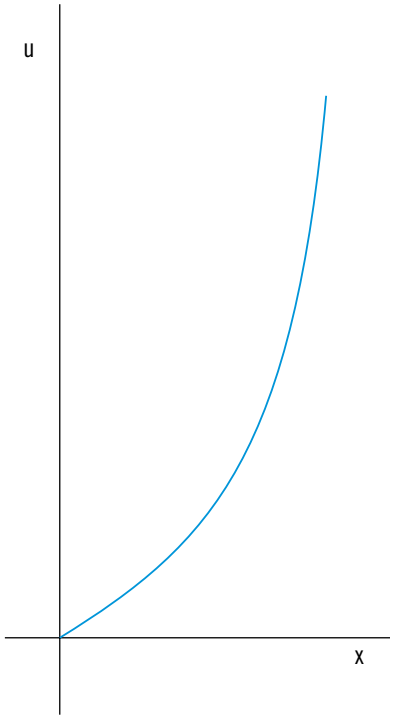
Finally I'm ready to make good on the second half of my two-part promise: the first part of which was to develop an easy-to-understand perspective cor-

rect texture mapper, and the second to speed it up to interactive performance.

In our efforts to speed up the mapper the obvious question is, "Where is the code currently spending its time?" I ran a profiler on it and ended up with what I like to call the perfect profile—almost all our time (96%) is spent in one function, `DrawScanLine`. The nice thing about a profile like this is that your optimization work in that function, sometimes called the hotspot, is very highly leveraged. In other words, every little bit you speed up the hotspot increases the overall performance by a lot. It's not surprising that `DrawScanLine` is the culprit because it contains the pixel loop, but it's always good to check our assumptions and gather some real data.

Listing 1 gives us a closer look at `DrawScanLine` inner loop. In it, we see that the function is doing a divide and two multiplies per pixel to figure out the texture coordinates, a multiply to calculate the texture offset, and a few adds. The divide is probably the major sink here. Divides are much slower than multiplies on most processors, and multiplies are generally slower than adds. If we comment out the divide per pixel and do a linear interpolation between the left and right edge, the mapper performs seven times faster in my test (and of course looks totally wrong because there's no perspective correction). Obviously, getting rid of the divide helps a lot. However, to get rid of the divide and keep the same visual quality we need to know exactly what the divide is doing there in the first place.

**Figure 1. The Perspective Curve**




The divide is the crux of the perspective texture mapper. It takes the linear interpolations of 1/z, u/z, and v/z and turns them into the nonlinear curve that samples pixels close together when the polygon is near the eyepoint and samples farther apart when the polygon is distant. If the polygon is slanted so that it's close on one end and distant on the other, the divide smoothly moves from close to separated samples. Figure 1 shows a plot of screen x versus sampled u for a typical perspective mapped scanline. As x increases, u starts increasing slowly and then grows much quicker. Obviously, this data is from a polygon that's close at low values of x and distant at larger values.

This curve (it's different for each polygon and scanline, in general) is what makes perspective mapping look correct, so the trick is to approximate the curve without using a divide and without adversely affecting our visual quality.

**Listing 1. The Inner Loop**

```
while(Width-- > 0) {
    float Z = 1/OneOverZ;
    int U = UOverZ * Z + 0.5;
    int V = VOverZ * Z + 0.5;

    *(pDestBits++) = *(pTextureBits + U + (V * TextureDeltaScan));

    OneOverZ += Gradients.dOneOverZdX;
    UOverZ += Gradients.dUOverZdX;
    VOverZ += Gradients.dVOverZdX;
}
```

## The Big Three

When I say we want to approximate the perspective curve, I mean we want to use alternate equations that will produce the same—or very close to the same—output (the u value shown in Figure 1) for the same input (x, as shown in Figure 1), but hopefully will be more efficient than our algorithm with its divides.

Three approximations to the perspective texture mapping equations are commonly used: subdividing affine, quadratic, and lines-of-constant-z.

I'm going to talk about lines-of-constant-z first because it's very different from the other two. First, "lines-of-constant-z" is an awkward name. Some people choose to call it "free-direction texture mapping," which is a bit easier to say but not quite as precise. Basically, a lines-of-constant-z rasterizer tries to take advantage of the neat fact that there are straight and parallel lines that have a constant z through any planar polygon. Imagine you're sliding a plane that's perpendicular to the z-axis back through your polygon. The plane will slice the polygon in a series of parallel lines as it moves through the depth range occupied by the polygon. All the pixels along one of these lines will have the z value of the plane, so the line has a "constant z."

Now, if you look at the math behind our projection, when z is constant, our equation turns into a linear interpolation (the perspective curve from one point on this line to the next is a line itself), which is quick and easy to compute and has no divides in the pixel loop. The down side is that in general you'll be interpolating along a diagonal line in the destination instead of across a nice horizontal scanline

(walls and floors are special cases where the lines-of-constant-z are vertical and horizontal, respectively).

This diagonal (or free-direction) interpolation causes three major problems. First, if your diagonal lines don't abut properly you'll get dropouts and overwrites inside your polygon. This is solvable if you're careful.

Second, unless you obey a strict fill convention, you'll get the same kind of dropouts and overwrites between polygons. This is much harder to fix than the intra-polygon problems, but it's still solvable.

Finally, and this one is the kiss of death as far as I'm concerned, it is totally impossible to achieve subpixel accuracy with a lines-of-constant-z texture mapper. To achieve subpixel accuracy we must always sample the texture from the pixel centers of the destination but our arbitrary line-of-constant-z doesn't hit the pixel center in the destination. However, you can't step off the line-of-constant-z to the pixel center or you'll need to divide to take the nonconstant-z step into account. Damned if you do, damned if you don't.

If you don't care about subpixel accuracy (insert flame about jittering and sloppy textures here), the lines-of-constant-z technique might be for you. As an added bonus, you get depth cuing effects like fog almost for free—you already know the depth of the current line, and the depth, by definition, is going to stay constant. So you can compute your fog value at the start and use it along the entire line instead of at every pixel.

The next two techniques, subdividing affine and quadratic, are based on a more straightforward approximation of the perspective curve. Both try to fit easy-to-interpolate curves to the more complex perspective curve. Subdividing affine does a piecewise linear approximation, fitting a number of line segments to the curve, and quadratic fits a quadratic curve to the perspective curve.

We're going to use a subdividing affine curve for our approximation, so before going into detail on it I'll go over the quadratic technique.

## Quadratics

Most people remember quadratic equations as parabolas from algebra. A quadratic in x is:

$$f(x) = ax^2 + bx + c \quad (1)$$

This equation will graph a parabola or a line on the x and f(x) axes. The coefficients a, b, and c determine the shape and position of the parabola on the graph, and we use these three "degrees of freedom" to attempt to fit a parabola to the perspective curve. We use the normal perspective mapper to interpolate down the edges of the polygon so they'll be precise, and use the quadratic to interpolate across the scanline (where we're spending our time in `DrawScanLine`). At each pixel on the scanline, we want to be able to feed in our screen position, x, and the quadratic equation should produce our texture coordinate as its value (f(x) in Equation 1). We need two quadratics—one to produce the u texture coordinate from x, and the other to produce the v coordinate from x.

> If you don't care about subpixel accuracy, the lines-of-constant-z technique might be for you.

Because we have three degrees of freedom, we can choose to match exactly any three characteristics of the perspective curve, and approximate the other characteristics. For example, we might choose to exactly interpolate the two endpoints, and spend our last degree of freedom on exactly matching the first derivative, or slope, of the perspective curve as it enters or leaves one of the endpoints—we can't match both derivatives because that would cost us two degrees of freedom, one more than we have left if we're going to hit the endpoints exactly. We could even interpolate one endpoint and exactly match the first and second derivative exactly at that point. Or we might spend all three degrees of freedom interpolating three points on the curve exactly, like the two endpoints and the middle point. We'll do the derivation for the latter approximation to show how it's done.

To figure out the quadratic curve that interpolates the two endpoints and the midpoint and produces the u texture coordinate given x, we start by writing down the equations we know. We assume x goes from 0 to 1 for simplicity. We need to solve for u at x = 0, 0.5, and 1 using the perspective divide to give us three known values, $u_0$, $u_1$, and $u_2$, respectively, so we can solve for a, b, and c, the three unknowns. We plug these values into Equation 1 to give us three equations in three unknowns:

$$f(0) = u_0 = a0^2 + b0 + c = c$$

$$f\left(\frac{1}{2}\right) = u_1 = a\left(\frac{1}{2}\right)^2 + b\frac{1}{2} + c = \frac{a}{4} + \frac{b}{2} + c$$

and:

$$f(1) = u_2 = a1^2 + b1 + c = a + b + c$$

We then solve the simultaneous equations for a, b, and c in terms of $u_0$, $u_1$, and $u_2$, and after a bit of algebra we get:

$$a = 2u_0 - 4u_1 + 2u_2$$

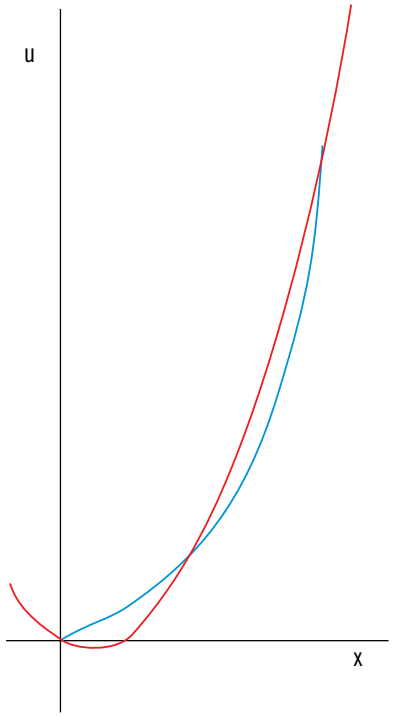
$$b = -3u_0 + 4u_1 - u_2$$

and:

$$c = u_0$$

Now, if we wanted to, we could use these coefficients and solve the quadratic directly at each point on the scanline (we'd actually do the math using x = 0, at x = width/2, and x = width so we wouldn't have to scale our x values between 0 and 1), but that means we'd be doing a bunch of multiplies per pixel—probably better than the divide we're currently doing, but not great. However, we can use forward differences to solve the quadratic using only addition and the solution for the previous pixel. Forward differences are covered in any good graphics or math textbook, but, simply stated, you take f(x+1)-f(x) to calculate the function's step based on its previous value. In the case of our qua-

**Figure 2. The Quadratic Curve**




**Listing 2. The Subdividing Affine `DrawScanLine` (Continued on p. 24)**

```
void DrawScanLine_suba( dib_info const &Dest,
    gradients_fx_fl_a const &Gradients,
    edge_fx_fl_a *pLeft, edge_fx_fl_a *pRight,
    dib_info const &Texture )
{
    int XStart = pLeft->X;
    int Width = pRight->X - XStart;

    char unsigned *pDestBits = Dest.pBits;
    char unsigned * const pTextureBits = Texture.pBits;
    pDestBits += pLeft->Y * Dest.DeltaScan + XStart;
    long TextureDeltaScan = Texture.DeltaScan;

    int const AffineLength = 8;

    float OneOverZLeft = pLeft->OneOverZ;
    float UOverZLeft = pLeft->UOverZ;
    float VOverZLeft = pLeft->VOverZ;

    float dOneOverZdXAff = Gradients.dOneOverZdX * AffineLength;
    float dUOverZdXAff = Gradients.dUOverZdX * AffineLength;
    float dVOverZdXAff = Gradients.dVOverZdX * AffineLength;

    float OneOverZRight = OneOverZLeft + dOneOverZdXAff;
    float UOverZRight = UOverZLeft + dUOverZdXAff;
    float VOverZRight = VOverZLeft + dVOverZdXAff;

    float ZLeft = 1/OneOverZLeft;
    float ULeft = ZLeft * UOverZLeft;
    float VLeft = ZLeft * VOverZLeft;

    float ZRight, URight, VRight;
    fixed16_16 U, V, DeltaU, DeltaV;

    if(Width > 0) {
        int Subdivisions = Width / AffineLength;
        int WidthModLength = Width % AffineLength;

        if(!WidthModLength) {
            Subdivisions--;
            WidthModLength = AffineLength;
        }

        while(Subdivisions-- > 0) {
            ZRight = 1/OneOverZRight;
            URight = ZRight * UOverZRight;
            VRight = ZRight * VOverZRight;

            U = FloatToFixed16_16(ULeft) + Gradients.dUdXModifier;
            V = FloatToFixed16_16(VLeft) + Gradients.dVdXModifier;
            DeltaU =
                FloatToFixed16_16(URight - ULeft)/AffineLength;
            DeltaV =
                FloatToFixed16_16(VRight - VLeft)/AffineLength;

            for(int Counter = 0;Counter < AffineLength;Counter++){
                int UInt = U>>16;
                int VInt = V>>16;

                *(pDestBits++) = *(pTextureBits + UInt +
                       (VInt * TextureDeltaScan));

                U += DeltaU;
                V += DeltaV;
            }

            ZLeft = ZRight;
            ULeft = URight;
            VLeft = VRight;
```

**Listing 2. The Subdividing Affine `DrawScanLine` (Continued from p. 22)**

```
            OneOverZRight += dOneOverZdXAff;
            UOverZRight += dUOverZdXAff;
            VOverZRight += dVOverZdXAff;
        }

        if(WidthModLength) {
            ZRight = 1/(pRight->OneOverZ - Gradients.dOneOverZdX);
            URight = ZRight *
                (pRight->UOverZ - Gradients.dUOverZdX);
            VRight = ZRight *
                (pRight->VOverZ - Gradients.dVOverZdX);

            U = FloatToFixed16_16(ULeft) + Gradients.dUdXModifier;
            V = FloatToFixed16_16(VLeft) + Gradients.dVdXModifier;

            if(--WidthModLength) {
                // guard against div-by-0 for 1 pixel lines
                DeltaU =
                    FloatToFixed16_16(URight - Uleft)
                    / WidthModLength;
                DeltaV =
                    FloatToFixed16_16(VRight - Vleft)
                    / WidthModLength;
            }

            for(int Counter = 0;
                Counter <= WidthModLength;Counter++) {
                int UInt = U>>16;
                int VInt = V>>16;

                *(pDestBits++) = *(pTextureBits + UInt +
                        (VInt * TextureDeltaScan));

                U += DeltaU;
                V += DeltaV;
            }
        }
    }
}
```

dratic, we need to do "second forward differences," where we calculate the forward difference of the function step. In other words, we calculate the forward difference of the forward difference.

How does it look? Well, in Figure 2, the blue curve is again the perspective curve, and the red curve is the quadratic interpolating the start, middle, and end. This is a pretty bad case for the quadratic because the perspective warp is quite high. On less distorted views the single quadratic would match up better. You can also subdivide into multiple quadratics to better match the curve if you want to spend the extra setup time. However, I chose this view because it illustrates a very significant side effect of the quadratic approximation—undershoot. If you look very closely, you'll see the red line actually dips below u = 0, which means we'd read off our texture map and possibly crash. You can figure out when this will happen and prevent it by subdividing, but that means even more setup in addition to setting up the quadratic equation for both u and v for each scanline.

Overall, the quadratic approximation is very elegant conceptually, but the problems of under- and overshoot and the setup overhead of calculating the coefficients seem to make it not worth the trouble. You can also use higher order curves, like cubics, and the math we've looked at extends easily. Perhaps we'll return to quadratics in a later column and see what we can do with them, but for now, we'll move on to subdividing affine.

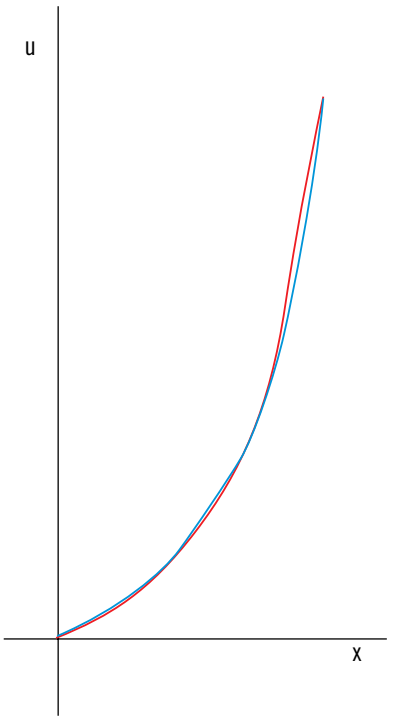
## It's Affine Day

It happens that the method we've chosen is also the simplest. You've probably already guessed exactly how subdividing affine texture mappers work from what I've been describing.

Basically, you solve the real perspective equation at a bunch of points along the scanline, and do a linear interpolation between those correct points (affine and linear are virtually interchangeable in this context). Linear interpolations are what we've been doing all along with 1/z, u/z, and v/z, so I won't go into detail here. Linear interpolations are very fast, and the setup isn't too bad for each affine span. The only trick is to determine how often to subdivide; the more you subdivide, the closer your piecewise linear curve will match the real curve—but the more overhead you'll have from divides and per-span setup.

One simple way to subdivide is to just break up the scanline into equisized spans. You can also adaptively subdivide based on the amount of perspective warp on each span. This issue's texture mapper will always subdivide to eight pixel spans, but we'll look into adaptive subdivision next time. Figure 3 shows a subdividing affine approximation to our

**Figure 3. The Subdivided Affine Curve**



favorite perspective curve, subdividing every eight pixels in x.

There are a few nice things about subdividing affine texture mappers. First, you can tune the performance and quality by setting the subdivision level. This lets you adjust your performance on demand (which you might need to do at runtime, depending on scene complexity).

Second, you can pretty easily fit all your interpolants in registers for the affine spans (a subject I'll cover in more depth in my next column).

Finally, unlike quadratic approximation, you'll never under- or overshoot with subdividing affine because like the real perspective curve, the affine spans are monotonically increasing or decreasing with the curve. In other words, depending on your subdivision granularity and the perspective warp, you'll sometimes draw the wrong pixels, but you'll never fetch outside the texture map or even outside the extents of the original correct span in texture space.

## Sample Code

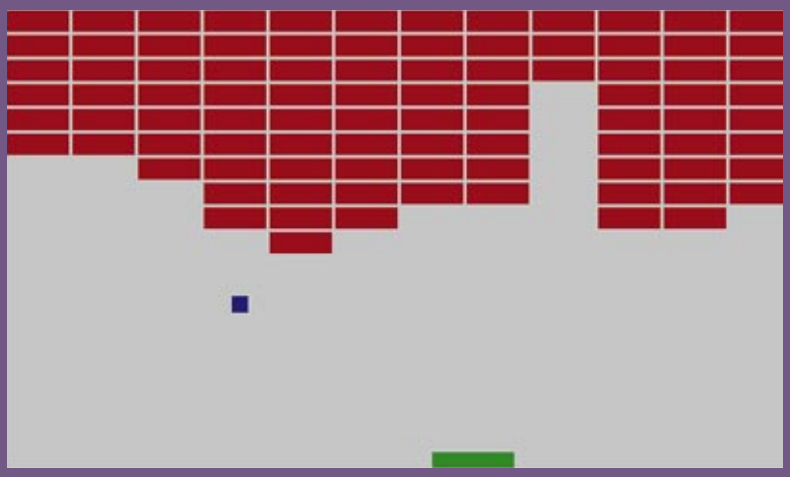
Listing 2 shows the `DrawScanLine` function modified to do subdividing affine texture mapping. We'll max this out next time, but even unoptimized it outperforms the divide-per-pixel routine by two to four times. We must treat the last span with care to ensure we interpolate the rightmost pixel correctly. You'll remember from previous articles that our right edge is actually the left edge of the next polygon over, so we need to subtract one pixel from the right edge to figure out the last pixel in *our* polygon and use it in the interpolation.

I've also finally written a texture mapping test bed so you can simply compile and run the listings. You can find it on ftp://ftp.mfi.com/gdmag. The test has all the texture mappers we've written so far, so you can see the jitter from the integer mapper, the mapping bug from the one we discussed in the last installment, and so on. It's a Windows program, but the code for the texture mappers is portable, and you'll be able to see exactly how they're called. The test bed is easy to modify—see the readme.txt file in the archive.

In parting, I want to mention a few tidbits. First, you might think you should do an approximation down the edges as well, which is certainly possible. However, your error can build up pretty quickly if you're not careful. Also, *Digital Image Warping* by George Wolberg (IEEE Computer Society, 1990) is a pretty good reference for this sort of thing (including forward differences). Finally, I'd like to thank Chris Green from Leaping Lizard Software for opening my eyes up to the fact that a, b, and c really are three totally arbitrary degrees of freedom. ■

*By the time you read this, Chris Hecker will have quit working for the man and will be out on his own, finally paying for his own beverages. You can recommend your favorite drink at checker@bix.com.*

# XSplat Breaks Out



An XSplat Breakout game in progress on Windows 95. The same source code works equally well on the Macintosh.

In *Writing Down the Bones*, author Natalie Goldberg describes how, at a local fair, she set up a booth where she wrote and gave away five-minute poems. Each poem represented a momentary thought, a unique idea recorded on paper and passed along on the wind without a backwards glance.

In a similar vein, John Carmack and John Romero wrote games for Softdisk Publishing, a service that sent new games to its subscribers every month, before starting id Software. Every 30 days, the programmers had to come up with a complete and innovative concept, make it happen, release it, and move on.

I envy the experiences of Goldberg and the team at id, who learned to move on, to avoid dwelling on mistakes. I've always found project ends to be painful, whether I'm finishing a chunk of code, a piece of writing, or a baked lasagna. The moment you sign off and pass along something personal, when you freeze it in time and drop it into the hands of its consumers, you find yourself standing naked in front of the world, wishing you could take it all back and do it right.

This is a roundabout way of saying that even as I introduced XSplat last month and used it to create a simple cross-platform paint program, I knew it contained major mistakes. Unfortunately, I haven't mastered the Zen of letting go, and this is after all a series of articles. So we're going to make four quick evolutionary swipes at XSplat before jumping into some real game code.

You'll find all the code included or referred to in this article on the *Game Developer* ftp site, ftp.mfi.com, in the /gdmag/src/ directory.

## Darwinism 101

Previously, I briefly mentioned virtualizing the `CXSplatWindow` member functions so we could use polymorphism in C++ to provide multiple window types in a single application using a single mechanism. The base `CXSplatWindow` class can provide a generic implementation from which we can derive any number of subclasses that can live together comfortably at run time. That's change number one: virtualization of `CXSplatWindow`.

Change number two involves sneaking around the much-touted event-based nature of our target operating systems. As it stands, XSplat also provides a strictly event-based system, meaning that it reacts to user input. However, a game must be proactive—the game should continue even when the user does nothing. So let's add a function `CXSplatWindow::Idle`, which the

system calls on every iteration of the main message loop, that we can use to process game logic. Hence, change number two: Idle Events.

Windows 95 came, and there was much rejoicing. The function provided by WinG under Windows 3.1 is now provided by the `CreateDIBSection` API. Chris Hecker went out of his way to make the WinG API identical to the `CreateDIBSection` API, so the two are nearly interchangeable. If you want to require Windows 95 or NT 3.5 and do away with WinG installation, you can use `CreateDIBSection`. That's change number three: Enable `CreateDIBSection`. We may revisit this later to detect the appropriate implementation at run time.

Finally, XSplat as we have it now lacks a useful keyboard interface, which we need to create keyboard-based games. Sending `KeyDown` messages alone doesn't quite cut it, so let's refine `CXSplatWindow::KeyDown` to notify us only when a key goes down (and not when a key autorepeats) and add `CXSplatWindow::KeyUp` to inform us when a key is released. That's change Number Four: Better Keyboard Events.

These quick tweaks prepare to move the XSplat base beyond a novelty item and towards a fully featured foundation for games. This excludes the `KeyDown` and `KeyUp` events, which require some explanation.

## Tuning the Keyboard

With `WM_KEYDOWN` and `WM_KEYUP` messages, Windows passes a virtual key code indicating the key that has been affected. We really want some more useful platform-independent key code, such as the ASCII value produced by the key in question, so we have to negotiate with Windows through a sequence of calls involving `GetKeyboardState` and `ToAscii`. If we succeed in getting a single ASCII code for the key, we can pass it on to the `CXSplatWindow` object. This precludes the use of special keys such as Control or Shift in XSplat games, but we'll make do.

The new `WM_KEYDOWN` handler, which replaces the old `WM_CHAR` handler, is shown in Listing 1. The `WM_KEYUP` handler is almost identical.

We also have to add some handling to the Macintosh message loop to process `keyUp` events. That's very straightforward, except for a little trick that caught me. The system engineers at Apple decided that most Macintosh applications don't listen to `keyUp` events and that they could cut some corners by not sending them, so you have to tell the Macintosh operating system explicitly to send `keyUp` events before it will pass along a single one. Be a good citizen and turn off `keyUp` notification when your application exits, too. The following code will enable this:

```
// Tell the OS to give us keyUp events
short OldEventMask = LMGetSysEvtMask();
LMSetSysEvtMask(OldEventMask |
  keyUpMask);
// Play the game!
XSplatMain();
// Restore the old system event mask
LMSetSysEvtMask(OldEventMask);
```

## Palettes and Static Cling

Another essential issue still remains untouched by XSplat: palettes and color management. In a multitasking, multiwindow environment, your application lives in harmony with many other applications

Jon Blossom

Before making XSplat faster, the cross-platform graphics library needs a little patching... Jon Blossom provides it in Part II of his five-part series.

## Listing 1. Translating a WM_KEYDOWN Message

```
case WM_KEYDOWN:
    // Only send KeyDown if the key is just going down
    // (no auto repeat)
    if (!(lParam & 0x40000000))
    {
        // We want the ASCII code of the key that´s going down
        // Begin with the key scan code, part of the virtual key
        UINT ScanCode = (lParam & 0x00FF0000) >> 16;

        // Wish we didn´t have to get all 256 entries, but...
        BYTE KeyState[256];
        GetKeyboardState(KeyState);

        // Convert the scan code to 1 or 2 ASCII characters
        char unsigned Key[2];
        int KeyCount =
            ToAscii(wParam, ScanCode, KeyState, (LPWORD)&Key, 0);

        // Only send the key if it translates to one ASCII char
        if (KeyCount == 1)
            pXSplatWindow->KeyDown((char unsigned)Key[0]);
    }
```

that require specific colors from the hardware, so both Macintosh and Windows include resident Palette Managers to arbitrate color requests. Understanding the Palette Managers will allow your XSplat applications to share this color sandbox with the other kids while still hoarding the resources you need.

On the Macintosh, you are entitled to explicitly program 254 of the 256 colors available in 8-bit video modes, the only modes with which XSplat deals. In the early days, all Macintosh systems had black-and-white monitors, so the original Macintosh operating system used only black and white for its user interface elements, and now the operating system is hard coded to program color 0 to White and color 255 to Black.

Giving two colors to the Macintosh system doesn't seem too painful, especially when most applications include black and white in their palettes anyway, so we won't worry much about the Macintosh Palette Manager. Windows, however, abides by much stricter rules. Read Ron Gery's very thorough online article "The Palette Manager: How and Why" (available on MSDN or through Microsoft Developer Relations group) for the full scoop. I'm going to give you the abridged version.

Back when VGA cards came onto the scene, Windows made the switch to color. At 640-by-480 pixels, the standard VGA allowed for 16 colors, which the Windows VGA driver programmed with 16 specific colors. Then along came video cards with a 256-color palette, but much of Windows had been coded assuming the 16 VGA colors of Windows 3.0. If Windows allowed programmers to tromp all over all the hardware colors, the colors used for window borders, captions, and other user interface elements could be replaced, and Windows's colorful user interface could be destroyed.

Windows programmers made a provision that would (under normal conditions) guarantee the availability of certain colors to draw the user interface. They took the 16 standard VGA colors, threw in a few new ones to spice things up, and came up with the "static colors" or the "cosmic colors," as one friend of mine calls them. The static colors are split between the first and last 10 entries in the palette so that a bitwise XOR of palette indices will also provide an XOR of the colors. In Windows 3.1, the values and number of static colors reserved depended on the video driver, but Windows 95 has standardized on 20 specific colors, listed in Table 1.

After the system claims its static colors, you usually have 246 left with which to work. You can tell the system to give up the static colors, leaving it only color 0 as black and color 255 as white, but I really recommend that only for full-screen applications that use none of the standard Windows user interface elements. In XSplat, we'll leave the static colors intact.

In fact, we'll go a step further. We'll actually include the Windows static colors in our Macintosh palettes to ensure that the colors available to XSplat applications won't depend on platform. The first and last 10 colors passed to the CXSplatWindow constructor will be always ignored, overwritten by the static colors.

Color 0 on the Macintosh is always white, while on Windows it's always black, no matter what we do. This "color zero problem" often shows up when artists hard-code an image to the palette on one system, expecting to be able to display the same image with the same 256-color palette on another system. When they bring the image from Macintosh to Windows, they find that all their whites have become black! Beware. An XSplat application should never use color 0 or color 255.

## Table 1. The Windows Static Colors

| Index | R-G-B Color | Index | R-G-B Color |
|---|---|---|---|
| 0 | 0x00 - 0x00 - 0x00 | 246 | 0XFF - 0xFB - 0xF0 |
| 1 | 0X80 - 0x00 - 0x00 | 247 | 0xA0 - 0xA0 - 0xA4 |
| 2 | 0x00 - 0x80 - 0x00 | 248 | 0x80 - 0x80 - 0x80 |
| 3 | 0x80 - 0x80 - 0x00 | 249 | 0xFF - 0x00 - 0x00 |
| 4 | 0x00 - 0x00 - 0x80 | 250 | 0x00 - 0xFF - 0x00 |
| 5 | 0x80 - 0x00 - 0x80 | 251 | 0xFF - 0xFF - 0x00 |
| 6 | 0x00 - 0x80 - 0x80 | 252 | 0x00 - 0x00 - 0xFF |
| 7 | 0xC0 - 0xC0 - 0xC0 | 253 | 0xFF - 0X00 - 0xFF |
| 8 | 0xC0 - 0xDC - 0xC0 | 254 | 0x00 - 0xFF - 0xFF |
| 9 | 0xA6 - 0xCA - 0xF0 | 255 | 0xFF - 0xFF - 0xFF |

### Identity Palettes

Our palette troubles aren't over yet. We may have figured out what colors we need in the palette, but there's still no guarantee that they'll appear in the system palette in the order we request them. Both operating systems go through complicated (read time-consuming) mapping algorithms to ensure that appropriate colors appear when you transfer an image from memory to the screen. If we manipulate the colors in the palette so that they exactly match the colors in the color tables of our offscreen images, the system will skip that mapping step, and copying an image to the screen will become a direct transfer from main memory to video memory, limited almost exclusively by the bandwidth of the bus between the two. We need to understand a few things before we can do this.

The Windows Palette Manager maintains a single system palette that holds all the colors actually available in the hardware, and every palletized application keeps a logical palette that represents the colors it has requested. The Palette Manager handles the mappings between the two. Basically, it uses three techniques to give you the colors you request, a process called "realization" of the palette.

The first technique is to prevent duplicate palette entries. If you request a color N that's already in the system palette at entry M, Windows will "collapse" your entry. When you request logical color N, it will actually use system color M. If you mark one of your entries PC_NOCOLLAPSE and the system palette isn't full, you can avoid this translation.

The second technique just serves your request by entering a new entry in the system palette, as long as the system palette isn't full. If there are M entries in the system palette, a new entry M will be created, and when you request logical entry N, you'll get system entry M.

The third technique is to map to the available colors. If you request a color N that's not in the system palette and all available system palette entries have been used, Windows will look for the color M in the system palette that best matches the requested color. If you mark a palette

## Listing 2. Windows Palette Creation

```
// A dummy LOGPALETTE structure
struct
{
    WORD Version;
    WORD NumberOfEntries;
    PALETTEENTRY aEntries[256];
} LogPalette = { 0x300, 256 };

// Grab the current palette for the display and count the
// number of static colors it´s using
HDC hdcScreen = GetDC(0);
int StaticColorCount = 20;
if (hdcScreen)
{
    StaticColorCount = GetDeviceCaps(hdcScreen, NUMCOLORS);
    GetSystemPaletteEntries(hdcScreen, 0, 256, LogPalette.aEntries);
    ReleaseDC(hdcScreen, 0);
}

// We´ll create our palette from the static colors,
// filling in whatever gaps are left with colors from
// the requested colors, or a gray wash as before if
// there are none.
int i;
for (i=0; i<StaticColorCount/2; ++i)
{
    // Fill in the peFlags of the first static entries
    LogPalette.aEntries[i].peFlags = 0;
}

if (Colors)
{
    // Fill in the middle entries with the requested colors
    // Count tells us where to find the appropriate RGB
    // triplet in the requested color array
    int Count = i * 3;

    for (; i<256 - StaticColorCount/2; ++i)
    {
        LogPalette.aEntries[i].peRed = Colors[Count++];
        LogPalette.aEntries[i].peGreen = Colors[Count++];
        LogPalette.aEntries[i].peBlue = Colors[Count++];

        // Mark as PC_RESERVED to guarantee identity palette
        LogPalette.aEntries[i].peFlags = PC_RESERVED;
    }
}
else
{
    // Fill in the middle entries with a grey wash
    for (; i<256 - StaticColorCount/2; ++i)
    {
        LogPalette.aEntries[i].peRed =
        LogPalette.aEntries[i].peGreen =
        LogPalette.aEntries[i].peBlue = i;

        LogPalette.aEntries[i].peFlags = PC_RESERVED;
    }
}

for (; i<256; ++i)
{
    // Fill in the peFlags of the remaining static entries
    LogPalette.aEntries[i].peFlags = 0;
}

// Finally, create the palette!
Palette = CreatePalette((LOGPALETTE*)&LogPalette);
```

## Listing 3. Macintosh Palette Creation

```
// Set up the palette from the given Colors
WinPalette = NewPalette(256, 0, pmExplicit | pmAnimated, 0);
if (WinPalette)
{
    // To maintain compatibility with our Windows cousin, we´ll include the Windows static colors
    // in the palette. Of course, color 0 must be white and color 255 black
    char unsigned const WindowsColorsLow[] =
        { 0xFF,0xFF,0xFF,
            0x80,0x00,0x00,  0x00,0x80,0x00,  0x80,0x80,0x00,
            0x00,0x00,0x80,  0x80,0x00,0x80,  0x00,0x80,0x80,
            0xC0,0xC0,0xC0,  0xC0,0xDC,0xC0,  0xA6,0xCA,0xF0 };
    char unsigned const WindowsColorsHigh[] =
        { 0xFF,0xFB,0xF0,  0xA0,0xA0,0xA4,  0x80,0x80,0x80,
            0xFF,0x00,0x00,  0x00,0xFF,0x00,  0xFF,0xFF,0x00,
            0x00,0x00,0xFF,  0xFF,0x00,0xFF,  0x00,0xFF,0xFF,
            0x00,0x00,0x00 };

    // We´ll create our palette from these "static" colors, filling in whatever gaps are left
    // with colors from the requested colors, or a gray wash as before if there are none
    // Count tells us where to find the appropriate RGB triplet in the requested color array
    int Count = 0;
    RGBColor rgb;
    int i;
    for (i=0; i<10; ++i)
    {
        // Fill in the first ten entries
        rgb.red = (long)WindowsColorsLow[Count++] << 8;
        rgb.green = (long)WindowsColorsLow[Count++] << 8;
        rgb.blue = (long)WindowsColorsLow[Count++] << 8;

        SetEntryColor(WinPalette, i, &rgb);
    }

    if (Colors)
    {
        // Fill in the middle entries with the requested colors
        for (; i<246; ++i)
        {
            rgb.red = (long)Colors[Count++] << 8;
            rgb.green = (long)Colors[Count++] << 8;
            rgb.blue = (long)Colors[Count++] << 8;

            SetEntryColor(WinPalette, i, &rgb);
        }
    }
    else
    {
        // Fill in the middle entries with a grey wash
        for (; i<246; ++i)
        {
            rgb.red = rgb.green = rgb.blue = (long)i << 8;

            SetEntryColor(WinPalette, i, &rgb);
        }
    }

    Count = 0;
    for (; i<256; ++i)
    {
        // Fill in the remaining static entries
        rgb.red = (long)WindowsColorsHigh[Count++] << 8;
        rgb.green = (long)WindowsColorsHigh[Count++] << 8;
        rgb.blue = (long)WindowsColorsHigh[Count++] << 8;

        SetEntryColor(WinPalette, i, &rgb);
    }
    SetPalette(Window, WinPalette, FALSE);
}
```

request PC_RESERVED, Windows will not map subsequent color requests to that entry.

We want to trick the Palette Manager into providing a one-to-one match between the colors we request and the colors in the system palette. In other words, we don't want our colors to be collapsed using the first technique, and we don't want any subsequent color requests to collapse into ours. Because PC_RESERVED is a superset of PC_NOCOLLAPSE, marking every one of our entries PC_RESERVED will do this for us.

So as long as we include the static colors and mark the rest PC_RESERVED, we can guarantee a one-to-one match of colors, but we may not yet match indices exactly. We're forcing the Palette Manager to use the second mapping technique, which means creating a new entry for every color we request, but it won't necessarily start creating new entries at color zero. If another application has requested palette entries before ours, our palette will be realized beginning where the last application left off, so M and N in the second technique may not be equal.

To remedy this, the WinG SDK included a sample function called ClearSystemPalette that forces the Palette Manager to realize a full palette of 256 PC_RESERVED colors, filling up the system palette so that subsequent palette mapping begins again at entry 0. Calling ClearSystemPalette in our XSplat WinMain and marking our logical palette entries PC_RESERVED will get us the pure identity mapping we want.

The Macintosh Palette Manager works almost the same way, collapsing entries when it is able to serve color requests as best it can. The Windows PC_NOCOLLAPSE corresponds loosely to the Macintosh pmExplicit flag, and the Windows PC_RESERVED corresponds to the Macintosh pmAnimated flag. To achieve an identity palette on the Macintosh, then, we'll put white at color 0 and black at color 255, and mark all other entries as pmExplicit | pmAnimated. There's no need for a ClearSystemPalette equivalent.

Listing 2 shows the code snippet that builds an identity palette as part of the Windows CXSplatWindow constructor. Listing 3 shows the equivalent code for the

Macintosh. The 20 static colors are hard coded for the Macintosh, but they're pulled out of the system palette under Windows. Also note the striking similarities between the code for the two platforms...

## Breaking Out

Enough of that system stuff, let's get to the real point: building games. We're going to begin programming a cross-platform version of an old classic, Breakout. You all know the game. A wall of bricks fills the upper half of the screen, and the player moves a paddle back and forth at the bottom of the screen to keep a ball bouncing into the bricks. Whenever the ball hits a brick, that brick goes away. The game ends when all the bricks have been destroyed.

XSplat now has everything we need to implement this game in full color: a place to create and display 256-color graphics (palletized `CXSplatWindow` and `COffscreenBuffer`), time to process the flow of the game (`Idle`), and user input to control the paddle (`KeyDown` and `KeyUp`). We can derive our game window, `CBreakoutWindow`, from the virtualized `CXSplatWindow` base and override the pieces we need to make the game work.

A `CBreakoutWindow` also keeps track of one complete game state. It stores the brick wall as an array of bytes, corresponding to the wall of bricks on the screen, and any nonzero value in the array indicates a brick in that location. When the edge of the ball crosses a brick boundary, the corresponding array entry is decremented. For now, `CBreakoutWindow::Initialize` initializes all entries with the value 1, so one hit of the ball will destroy a brick.

The ball and paddle both have position and velocities, also stored in the `CBreakoutWindow` object. The paddle is limited to motion along the X-axis while the ball can move along X and Y. On `KeyDown` and `KeyUp` events, we'll adjust the speed of the paddle, moving it left as long as the player holds down the < key, and right as long as the player holds down the > key. On `Idle` messages, the game logic kicks in, moving the ball and checking if it has hit anything important. Then `Idle` redraws the game, using simple rectangle drawing code I've described previously (see "XSplat: A Foundation for Cross-Platform Development," Oct./Nov. 1995), and copies it to the screen.

## Listing 4. Breakout Declarations

```
// Breakout Game Window
// The Breakout window is a regular XSplat window with
// most functions overridden. The game is played entirely
// within this window, so the Breakout window includes
// all state information necessary for the game.

class CBreakoutWindow : public CXSplatWindow
{
public:
    // Game setup occurs during construction and on
    // InitGame calls
    CBreakoutWindow(unsigned char const *Colors=0);
    void InitGame(void);

    // The destructor will clean up the game
    virtual ~CBreakoutWindow(void);

    // All of the game logic takes place on Idle
    virtual void Idle(void);

    // This game cares about key states
    virtual void KeyDown(char unsigned Key);
    virtual void KeyUp(char unsigned Key);

protected:
    // This function will completely redraw the game state
    void DrawCompleteGameState(void);

    // We´ll provide a ´demo mode´ for the game to play itself
    int IsDemoMode;

    // Game Data
    // Game element position and speed
    int BallX, BallY;
    int BallXSpeed, BallYSpeed;
    int PaddleX;
    int PaddleXSpeed;

    // Game field
    char unsigned GameField[kWallWidthBricks * kWallHeightBricks];

    inline char unsigned GetBrickState(int X, int Y)
        { return GameField[Y * kWallWidthBricks + X]; };

    inline void SetBrickState(int X, int Y, char unsigned State)
        { GameField[Y * kWallWidthBricks + X] = State; };

    inline char unsigned HitBrick(int X, int Y);
};

inline char unsigned CBreakoutWindow::HitBrick(int X, int Y)
{
    char unsigned ReturnState = 0;

    if (X >= 0 && Y >= 0 &&
         X < kWallWidthBricks &&
         Y < kWallHeightBricks)
    {
        int Index = Y * kWallWidthBricks + X;
        ReturnState = GameField[Index];
        if (ReturnState)
            GameField[Index]--;
    }
    return ReturnState;
}
```

Listing 4 shows the declaration of `CBreakoutWindow`, and Listing 5 shows the

## Listing 5. Breakout Game Implementation (Continued on p. 33)

```
void CBreakoutWindow::KeyDown(char unsigned Key)
{
    if (!IsDemoMode)
    {
        // Get the paddle moving
        switch (Key)
        {
            case ´,´:
            case ´<´:
                PaddleXSpeed -= 2;
                break;

            case ´.´:
            case ´>´:
                PaddleXSpeed += 2;
                break;

            case ´d´:
            case ´D´:
                // Switch to demo mode
                IsDemoMode = 1;
                break;

            default:
                break;
        }
    }
    else if (Key == ´P´ || Key == ´p´)
    {
        // Switch to player control
        IsDemoMode = 0;
    }
}

void CBreakoutWindow::KeyUp(char unsigned Key)
{
    if (!IsDemoMode)
    {
        // Reverse the motion of the paddle
        switch (Key)
        {
            case ´,´:
            case ´<´:
                PaddleXSpeed += 2;
                break;

            case ´.´:
            case ´>´:
                PaddleXSpeed -= 2;
                break;

            default:
                break;
        }
    }
}


//------------------------------
// Game logic is processed while the game is idle

void CBreakoutWindow::Idle(void)
{
    // Don´t do anything while backgrounded
    if (!IsActiveFlag)
        return;

    //------------------------------
    // Move the ball and calculate a bounce off any walls

    int BounceX = 0;
    int BounceY = 0;

    BallX += BallXSpeed;
    BallY += BallYSpeed;

    if (BallX < kPlayAreaLeft)
    {
        BallX = kPlayAreaLeft + (kPlayAreaLeft - BallX);
        BounceX = 1;
    }
    else if (BallX >= kPlayAreaRight - kBallSize)
    {
        BallX = 2*kPlayAreaRight - 2*kBallSize - BallX;
        BounceX = 1;
    }

    if (BallY < kPlayAreaTop)
    {
        BallY = kPlayAreaTop + (kPlayAreaTop - BallY);
        BounceY = 1;
    }

    //------------------------------
    // Follow the ball with the paddle center when in demo mode,
    // allow the user to control the paddle in play mode

    if (IsDemoMode)
    {
        if (PaddleX < BallX + kBallSize/2) ++PaddleX;
        else if (PaddleX > BallX + kBallSize/2) --PaddleX;
    }
    else
    {
        PaddleX += PaddleXSpeed;
    }

    // Make sure the paddle doesn´t move out of the play area!
    if (PaddleX < kPlayAreaLeft + kPaddleWidth/2)
        PaddleX = kPlayAreaLeft + kPaddleWidth/2;
    else if (PaddleX > kPlayAreaRight - kPaddleWidth/2)
        PaddleX = kPlayAreaRight - kPaddleWidth/2;

    //------------------------------
    // Bounce the ball

    if (BallY + kBallSize >= kWallTop && BallY < kWallBottom)
    {
        // Ball is within the brick field.
        // Calculate ball bounces off bricks
        int BrickLeft = (BallX -  kWallLeft) / kBrickWidth;
        int BrickTop = (BallY - kWallTop) / kBrickHeight;
        int BrickRight = (BallX + kBallSize -  kWallLeft) / kBrickWidth;
        int BrickBottom = (BallY + kBallSize - kWallTop) / kBrickHeight;

        // Look to the sides of the ball
        if (BallXSpeed > 0 &&
            ((BallX + kBallSize - kWallLeft) % kBrickWidth) == 0)
        {
            if (HitBrick(BrickRight, BrickTop))
                BounceX = 1;

            if (BrickTop != BrickBottom &&
                 HitBrick(BrickRight, BrickBottom))
                BounceX = 1;
        }
        else if (BallXSpeed < 0 &&
            ((BallX - kWallLeft) % kBrickWidth) == kBrickWidth-1)
```

## Listing 5. Breakout Game Implementation (Continued from p. 32)

```
    {
        if (HitBrick(BrickLeft, BrickTop))
            BounceX = 1;

        if (BrickTop != BrickBottom &&
             HitBrick(BrickLeft, BrickBottom))
            BounceX = 1;
    }

    // Look to the top and bottom of the ball
    if (BallYSpeed > 0 &&
        ((BallY + kBallSize - kWallTop) % kBrickHeight) == 0)
    {
        if (HitBrick(BrickLeft, BrickBottom))
            BounceY = 1;

        if (BrickLeft != BrickRight &&
             HitBrick(BrickRight, BrickBottom))
            BounceY = 1;
    }
    else if (BallYSpeed < 0 &&
        ((BallY - kWallTop) % kBrickHeight) == kBrickHeight-1)
    {
        if (HitBrick(BrickLeft, BrickTop))
            BounceY = 1;

        if (BrickLeft != BrickRight &&
             HitBrick(BrickRight, BrickTop))
            BounceY = 1;
    }
}
else if (BallY == kPlayAreaBottom - kPaddleHeight - kBallSize)
{
    // The ball is at the correct height to bounce off the paddle
    if (BallX + kBallSize > PaddleX - kPaddleWidth/2 &&
        BallX <= PaddleX + kPaddleWidth/2)
    {
        // The ball hit the paddle, so it´s going to bounce back up
        BounceY = 1;

        // Divide the paddle into five sections and determine which
        // contains the ball
        int BallXCenter = BallX + kBallSize/2;
        int PaddleLeft = PaddleX - kPaddleWidth/2;
        int PaddleSection = (BallXCenter - PaddleLeft)
            / (kPaddleWidth / 5);

        if (BallXSpeed == 0 && PaddleSection != 2)
        {
            // If the ball isn´t moving in X and it hits outside of
            // the paddle center, it should bounce to the side
            if (PaddleSection < 2)
                BallXSpeed = -1;
            else
                BallXSpeed = 1;
        }
        else if (PaddleSection == 0 && BallXSpeed > 0)
        {
            // The ball is striking the leftmost section of
the paddle
            // from the left. Bounce it back.
            BounceX = 1;
        }
        else if (PaddleSection == 4 && BallXSpeed < 0)
        {
            // The ball is striking the rightmost section of
the paddle
            // from the right. Bounce it back.
            BounceX = 1;
        }
        else if (PaddleSection == 2)
        {
            // When it hits in the very center section, the
ball
            // bounces straight up.
            BallXSpeed = 0;
        }
    }
}
else if (BallY == kPlayAreaBottom - kBallSize)
{
    // TODO: Lose ball!
    BounceY = 1;
}

//------------------------------
// Perform any velocity changes due to bounces

if (BounceX) BallXSpeed = -BallXSpeed;
if (BounceY) BallYSpeed = -BallYSpeed;

//------------------------------
// Display the new frame and return

COffscreenBuffer* pBuffer = GetOffscreenBuffer();
if (pBuffer)
{
    pBuffer->Lock();
    DrawCompleteGameState();
    pBuffer->SwapBuffer();
    pBuffer->Unlock();
}
}
```

implementation of relevant `CBreakoutWindow` functions. Many of these functions use constants (beginning with a "k") that should be self-explanatory. Notice that there's absolutely nothing platform-specific in the game code; it's all built on XSplat and compiles for Windows and Macintosh without change.

Again, I encourage you to grab the complete source code and compiled XSplat Breakout applications from the *Game Developer* ftp site.

### Next Time

While the version of Breakout built in this article works and is even fun to play for about a minute, it certainly has a long way to go. For one thing, it's running flat out all the time, and it hugs a root even on my Pentium 90. Perhaps that has something to do with the fact that we're redrawing the entire screen every frame?

In my next article, we'll streamline the Breakout rendering by adding a `SwapRect` function to `COffscreenBuffer`, and we'll take a look some platform-independent timing functions that will allow us to control the final speed of the game. And, of course, we'll start spicing up game play with some extra colors, increasing speed, and increasing difficulty levels.

Until then, have fun! ■

*You can reach Jon Blossom via e-mail at blossom@mobius.net or through* Game Developer *magazine.*

# Under the Rainbow


Two examples of color quantization in action. On the left, the cover of a magazine you might recognize, scanned at 16 million colors. On the right, the original scan reduced to 20 colors with full variance minimization.

Pick a color, any color. When I was younger, my brother, who enjoyed magic and card tricks, would frequently use me to beta test his sleight of hand, and he had a wisecracking way of saying, "Pick a card, any card." Of course, all cards were not created equal, and I always tried to guess which card would throw a monkey wrench into his trick, as brothers are wont to do. Now after working on image processing products for my start-up company, I'm faced with similar choices in color quantization algorithms.

I can already hear some of you saying, "Wait a second, didn't we do this a year ago?" In a word, yes we did ("A Few Good Colors," December 1994). So why are we revisiting this process? The answer's simple: There is a newer and better algorithm available called variance minimization. More specifically, I want to present to you an implementation of this algorithm that is fast, practical, and produces visibly better results in circumstances common in game development.

This article has two sections. First, we are going to discuss how to manage colors and graphics in the game development process and look at issues in getting artwork into your program. Once we have examined the situations where quantization is needed, we are going to get down to the nuts and bolts of variance minimization, and explain why it gets results that are superior to the commonly used Heckbert statistical and popularity methods.

## Color Usage in Games

First, I asked the question "Does color quantization have a place in game development?" I found that until about six or seven years ago the answer was, "Not really." Since then, it's quickly gone from "Not really" to "Oh, yes." The advent of 256-color graphics modes on PCs and Macintoshes spurred this change of tone. Prior to that, colors were so limited that game developers got their graphics from skilled artists who hand-drew each image. At the time, no popular display could show more than 16 of 512 set colors, and PCs were limited to 64 possible colors at best. With the shift from digital to analog color monitors came video cards that could display 256 simultaneous colors, which was a great improvement. More impor-

tantly, these video cards were capable of drawing those colors from a range of a quarter million to 16 million colors—and for the first time the cards could respectably display photographic or video images.

## Color and Palette Management Issues

Let's talk about artwork and graphics for a minute. The graphics for a typical game can come from any combination of the following (each of which raises issues that can affect the quality of the finished product):

- Traditional drawn images, which are scanned.
- Images drawn with a computer paint program.
- Computer-generated images such as raytraced images.
- Images created with a video camera and video capture board.

Having an artist paint or draw images to be scanned is best suited for larger images, such as backgrounds, or small images with a photographic look, such as faces of characters. Getting the artwork into the program involves scanning, resampling (resizing), quantization, and possibly dithering. Ideally, you want to make this a one-step process. Whenever an intermediate image file exists, some visual information will be lost.

To get the best final results, follow two steps. Scan the image at the highest physical (not interpolated) resolution and store it in a lossless true color (24-bit) format such as TIFF. Check the scanned image at this point to see if the color and contrast closely match the image and adjust and rescan until you are satisfied. You now have the electronic equivalent of a photograph negative. From this file you can produce the graphics that will actually go into your program. If you make changes to the number of colors used or want to make derivative images (tinted, inverted, brightened, and so on), going back to the original scan file lets you maintain the highest quality.

Drawing images with a computer paint program has some benefits, but provides some new wrinkles. With a paint program, you can eliminate the need to resize an image. If a graphic needs to be a specific size, the artist just sets the work area to that size and begins drawing. Because the artist draws with pixels instead of pen or brush, he or she has precise control of the final result. So the artist can draw lines perfectly straight, make distances between objects precise, and make color gradients perfectly uniform. When it comes to small images or objects, a paint program is often the way to go.

Artists often use video capture equipment when they want multiple angles or positions of the same scene or object. Video data usually uses the YUV or HSV (Hue, Saturation and Brightness) color model and is converted by the hardware or driver software into RGB colors.

## The Palette is a Resource

Another issue in game development is allocating your colors for each screen or situation. Imagine that you are producing a fighting game for the PC, much like the "Super Mortal Killer Turbo

Matt Pritchard and
Rich Geldreich Jr.

**Game graphics are nothing without true, vibrant color that doesn't sap memory. Try the high-speed algorithm offered here and you'll optimize color quantization in your game design.**

Dragon II" games you see in the arcades. At any given time, you have several independent images on the screen—the background, the players, and their weapons. As the game progresses, these images are constantly replaced with different images. Odds are that all those graphics started out as some sort of true color image and were quantized to use a small number of discrete colors. Think about what is going on with the palette in such a game. There are two basic ways to handle the palette.

First, the game might use a global, fixed palette that multiple images draw from. This lets each image use a larger number of different colors, but because they must be shared with all other graphics, the quantization error—the degree to which the displayed color differs from the original color—will tend to be much greater.

The other approach is to use a segmented palette, where each object gets an exclusive portion of the palette for its own use. This gives the object fewer colors to use, but allows the quantization error to be kept to a minimum. This method allows for a much larger total number of colors to be used.

It turns out that selecting colors specifically for an image (quantizing for that image alone) usually more than makes up for the reduction in available colors. It also turns out that the further you reduce the number of colors, the more impact the quantization algorithm has. (Dithering also has a place here, but that is a topic for a separate article.)

Two more palette management issues often show up in game development—palette effects and shading. Palette effects are just that: special effects, usually in the form of cycling, pulsating or flashing colors. No big deal, except that when you add palette effects, you usually have to reserve specific portions of the palette for them. Shading, on the other hand, often involves splitting the palette up into sections for each brightness level.

So where are we going with all this talk about color, artwork, and graphics issues? This is simply an illustration of all the ways and situations in which you need to quantize images to an arbitrary number (usually a small one) of colors before they go into your final product.

## Supporting Multiple Platforms

I want to mention one more situation: porting your game and graphics to multiple platforms. On a PC, under DOS you have full control of all 256 entries in the palette. But what if you are doing a Windows version? If you have never programmed for Windows before, you are in for a rude surprise. In 256-color modes, Windows reserves 20 colors for its own use. It is possible to get 18 of those back, but only at the cost of screwing up all the colors in the desktop. The new Microsoft Game SDK provides support for taking over the screen, but what if your program needs be windowable on the desktop? You are looking at a practical maximum of 236 colors and you have two choices.

You can let Windows manage the colors for you, and it will translate some of your graphics colors into other colors at the expense of speed and image quality. If you use 256-color images for your wallpaper, you've probably already seen this in action. When a program needs some additional colors, parts of the wallpaper flash for an instant before they are replaced with what Windows considers the most similar colors that it can spare. A better option is to redo your 256-color program to work with 236 (or fewer) colors.

But why stop with Windows? What about OS/2 or the Macintosh? And if your product is a successful game, you could wind up porting it to dedicated game systems such as Sony, Nintendo, Atari, or Sega, each with different color systems.

## The Example Program and Source

Now that I've built the case for quantization, it is time to deliver the goods. Taking code directly from our PC image viewer, PowerView, we have created a program that takes the raw output of PicLab, a public domain image processing Program by Lee Daniel Crocker, and quantizes it to an arbitrary number of colors from 2 to 256. Unfortunately, the 50K of Watcom C source code is too cumbersome to list completely here. However, the complete source, compiled program, and example files are freely available on the *Game Developer* ftp site, ftp.mfi.com, and in our CompuServe library (GO SDFORUM), in the archive file Dec95.zip, subfile VARMINCQ.zip. You can also access the shareware version of PowerView here, in the archive file PVIEW100.zip

## The Quantization Process

From here on, we are going to assume some familiarity with other quantization methods that use cube splitting. Variance minimization is also a cube-splitting process, but it differs from the more common Heckbert method in several important ways.

So far, the only published information on variance minimization is a paper by Xiaolin Wu, ("Efficient Statistical Computations for Optimal Color Quantization,"*Graphics Gems II*, Academic Press Inc., 1991). Upon examination, we found the source code Wu presented with his paper was not practical on a PC class system. So we threw out his source and developed our own highly efficient implementation, which we present here. Wu's paper is a very academic read and will send average programmers scrambling for their college calculus books. So, we'll focus on the actual implementation and how it differs from other quantization methods. If you're interested in the theory and math, by all means check out Wu's paper.

For those not familiar with how computers process color, here's a quick explanation. Every dot of color that appears on your monitor is built from three separate component colors, red, green, and blue. Each component has a brightness value that can range from none (black) to the maximum intensity of the display device. Your computer's video card uses a numerical value to represent the brightness of each color component.

The computer industry has settled

## Listing 1. Structure Used for a Colorspace Box

```
   typedef struct
{
uint variance;         /* weighted variance */
    uint total_weight; /* total weight */
    uint tt_sum;       /* tt_sum += r*r+g*g+b*b*weight over entire box */

    uint t_ur;         /* t_ur += r*weight over entire box */
    uint t_ug;         /* t_ug += g*weight over entire box */
    uint t_ub;         /* t_ub += b*weight over entire box */

    int  ir, ig, ib;   /* upper and lower bounds */
    int  jr, jg, jb;
} box;
```

on 8-bit (one-byte) numbers, which allows for 256 possible settings between black and maximum intensity for each color component. With three components, the number of possible colors is 256 by 256 by 256 or 16.8 million. That's a lot of colors! It also gives us a way to compare colors. We can create a three-dimensional cube, and let each axis (X,Y, and Z) represent one of the color components (R, G, and B). This way, each color can be mapped to a point inside the cube by using its RGB values as the X,Y, and Z coordinates of the color point. Now that we can represent each color as a point in three-dimensional space, we can use simple geometry to compute how close together any two colors are. Using the Pythagorean theorem, we define the distance between color 1 (r1,g1,b1) and color 2 (r2,b2,g2) as the square root of $(r1-r2)^2 + (g1-g2)^2 + (b1-b2)^2$. Figure 1 illustrates the RGB three-dimensional color space cube and how to locate a specific RGB color in it.

Quantization is the process of replacing one color with another color; usually from a smaller selection of color points. The distance between the two colors is called the quantization error.

Our process starts with a color histogram of the image to be quantized. For those not familiar with the term, a color histogram is a three-dimensional array of the RGB colors used. The value of a specific element is the number of times the color with that element's RGB coordinates appears in the image. Typically, most elements in the histogram will be unused or zero. Almost every quantization routine I have ever seen reduces a 24-bit color to 15 bits when building the histogram, due to memory considerations.

Using short integers, a 15-bit histogram takes 64K of memory, while a 24-bit histogram would require 32 megabytes. This reduction (typically truncation of the low bits) is in itself a form of quantization that is actually noticeable. If you have ever seen an image of a blue sky that has curved bands as it approaches the horizon, you've seen the effects of this truncation. With the advent of 32-bit compilers, I have found it practical to use a 6-bit RGB histogram, which takes 512K of memory and produces superior results compared to the 5-bit histogram. (We are using two bytes per entry, while many implementations use four bytes per entry. With two bytes, remember you need to take precautions to keep from overflowing the histogram count.)

Once we have this three-dimensional color histogram, we put a box around it. The box structure is shown in Listing 1. Then, we shrink the box to the smallest possible box that can contain all of the used points within. In fact, every time we create or split a box, we shrink it immediately. Shrinking the box is an important optimization used in most quantization processes. It can create empty regions in the histogram that are not contained in any box and must be accounted for when building the inverse color map.

Now we get to the question, "What exactly is variance?" Consider that all the colors in a given box are going to be represented by a color point, also known as the "representative" color. Every used color in that box is going to be replaced by the representative color, and some degree of quantization error is going to be introduced into the image when those pixels are replaced with the quantized color. Variance in this case means the total quantization error of all the used points in the box, weighted by the usage count of each point.

## Listing 2. The Weighted Variance Calculation

```
static uint variance(uint tw, uint tt_sum,
       uint t_ur, uint t_ug, uint t_ub)
{
    double temp;

    temp  = (double)t_ur * (double)t_ur;
    temp += (double)t_ug * (double)t_ug;
    temp += (double)t_ub * (double)t_ub;
    temp /= (double)tw;

    return ((uint)((double)tt_sum - temp));
}
```

The goal of variance minimization is just what the name implies; to pick a representative color that results in the lowest possible total variance for that box. If you think about it, that's the goal of quantization as well—to make the color change in each pixel, (that is, the quantization error), as small as possible. The smaller the variance, the less change to the image. I've shown our variance calculation in Listing 2.

In this algorithm, we define the representative color as the "center of color gravity" for the box—that is, the

*(Continued on p. 40)*

## Listing 3. Splitting a Colorspace Box (Continued on p. 39)

```
/*---------------------------------------*/
/* Splits box along the axis which will
/* minimize the two new box´s overall
/* variance. A brute force search is used
/* to locate the optimum split point. */
/*---------------------------------------*/
static void split_box(box *old_box)
{
    int i, j;
    box *new_box;

    uint total_weight;
    uint tt_sum, t_ur, t_ug, t_ub;
    int  ir, ig, ib, jr, jg, jb;

    uint total_weight1;
    uint tt_sum1, t_ur1, t_ug1, t_ub1;
    int  ir1, ig1, ib1, jr1, jg1, jb1;

    uint total_weight2;
    uint tt_sum2, t_ur2, t_ug2, t_ub2;
    int  ir2, ig2, ib2, jr2, jg2, jb2;

    uint total_weight3;
    uint tt_sum3, t_ur3, t_ug3, t_ub3;

    uint lowest_variance, variance_r, variance_g, variance_b;
    int  pick_r, pick_g, pick_b;

    new_box = boxes + num_boxes;
    num_boxes++;

    total_weight    = old_box->total_weight;
    tt_sum  = old_box->tt_sum;
    t_ur        = old_box->t_ur;
    t_ug        = old_box->t_ug;
    t_ub        = old_box->t_ub;
    ir          = old_box->ir;
    ig          = old_box->ig;
    ib          = old_box->ib;
    jr          = old_box->jr;
    jg          = old_box->jg;
    jb          = old_box->jb;

    /* left box´s initial statistics */

    total_weight1   = 0;
    tt_sum1 = 0;
    t_ur1       = 0;
    t_ug1       = 0;
    t_ub1       = 0;

    /* right box´s initial statistics */

    total_weight2   = total_weight;
    tt_sum2 = tt_sum;
    t_ur2       = t_ur;
    t_ug2       = t_ug;
    t_ub2       = t_ub;

    /* locate optimum split point on red axis */

    variance_r = 0xFFFFFFFF;

    for (i = ir; i < jr; i++)
    {
        uint total_variance;

        /* calculate the statistics for the area being taken
         * away from the right box and given to the left box
         */

        sum(i, ig, ib, i, jg, jb,
            &total_weight3, &tt_sum3, &t_ur3, &t_ug3, &t_ub3);

#ifdef DEBUGGING
        if (total_weight3 > total_weight)
            ASSERT(TRUE)
#endif

        /* update left and right box´s statistics */

        total_weight1   += total_weight3;
        tt_sum1 += tt_sum3;
        t_ur1       += t_ur3;
        t_ug1       += t_ug3;
        t_ub1       += t_ub3;

        total_weight2   -= total_weight3;
        tt_sum2 -= tt_sum3;
        t_ur2       -= t_ur3;
        t_ug2       -= t_ug3;
        t_ub2       -= t_ub3;

#ifdef DEBUGGING
        if ((total_weight1 + total_weight2) != total_weight)
            ASSERT(TRUE)
#endif

        /* calculate left and right box´s overall variance */

        total_variance = variance(total_weight1, tt_sum1, t_ur1, t_ug1, t_ub1) +
                variance(total_weight2, tt_sum2, t_ur2, t_ug2, t_ub2);

        /* found better split point? if so, remember it */

        if (total_variance < variance_r)
        {
            variance_r = total_variance;
            pick_r = i;
        }
    }

    /* left box´s initial statistics */

    total_weight1   = 0;
    tt_sum1 = 0;
    t_ur1       = 0;
    t_ug1       = 0;
    t_ub1       = 0;

    /* right box´s initial statistics */

    total_weight2   = total_weight;
    tt_sum2 = tt_sum;
    t_ur2       = t_ur;
    t_ug2       = t_ug;
    t_ub2       = t_ub;

    /* locate optimum split point on green axis */

    variance_g = 0xFFFFFFFF;

    for (i = ig; i < jg; i++)
    {
        uint total_variance;

        /* calculate the statistics for the area being taken
```

## Listing 3. Splitting a Colorspace Box (Continued on p. 40)

```
        * away from the right box and given to the left box
        */

        sum(ir, i, ib, jr, i, jb,
            &total_weight3, &tt_sum3, &t_ur3, &t_ug3, &t_ub3);

#ifdef DEBUGGING
        if (total_weight3 > total_weight)
            ASSERT(TRUE)
#endif

        /* update left and right box's statistics */

        total_weight1   += total_weight3;
        tt_sum1 += tt_sum3;
        t_ur1       += t_ur3;
        t_ug1       += t_ug3;
        t_ub1       += t_ub3;

        total_weight2   -= total_weight3;
        tt_sum2 -= tt_sum3;
        t_ur2       -= t_ur3;
        t_ug2       -= t_ug3;
        t_ub2       -= t_ub3;

#ifdef DEBUGGING
        if ((total_weight1 + total_weight2) != total_weight)
            ASSERT(TRUE)
#endif

        /* calculate left and right box's overall variance */

        total_variance =
            variance(total_weight1, tt_sum1,
                t_ur1, t_ug1, t_ub1) +
            variance(total_weight2, tt_sum2,
                t_ur2, t_ug2, t_ub2);

        /* found better split point? if so, remember it */

        if (total_variance < variance_g)
        {
            variance_g = total_variance;
            pick_g = i;
        }
    }

    /* left box's initial statistics */

    total_weight1   = 0;
    tt_sum1 = 0;
    t_ur1       = 0;
    t_ug1       = 0;
    t_ub1       = 0;

    /* right box's initial statistics */

    total_weight2   = total_weight;
    tt_sum2 = tt_sum;
    t_ur2       = t_ur;
    t_ug2       = t_ug;
    t_ub2       = t_ub;

    variance_b = 0xFFFFFFFF;

    /* locate optimum split point on blue axis */

    for (i = ib; i < jb; i++)
    {
        uint total_variance;

        /* calculate the statistics for the area being taken
        * away from the right box and given to the left box
        */

        sum(ir, ig, i, jr, jg, i,
            &total_weight3, &tt_sum3, &t_ur3, &t_ug3, &t_ub3);

#ifdef DEBUGGING
        if (total_weight3 > total_weight)
            ASSERT(TRUE)
#endif

        /* update left and right box's statistics */

        total_weight1   += total_weight3;
        tt_sum1 += tt_sum3;
        t_ur1       += t_ur3;
        t_ug1       += t_ug3;
        t_ub1       += t_ub3;

        total_weight2   -= total_weight3;
        tt_sum2 -= tt_sum3;
        t_ur2       -= t_ur3;
        t_ug2       -= t_ug3;
        t_ub2       -= t_ub3;

#ifdef DEBUGGING
        if ((total_weight1 + total_weight2) != total_weight)
            ASSERT(TRUE)
#endif

        /* calculate left and right box's overall variance */

        total_variance =
            variance(total_weight1, tt_sum1,
                t_ur1, t_ug1, t_ub1) +
            variance(total_weight2, tt_sum2,
                t_ur2, t_ug2, t_ub2);

        /* found better split point? if so, remember it */

        if (total_variance < variance_b)
        {
            variance_b = total_variance;
            pick_b = i;
         }
    }

    /* now find out which axis should be split */

    lowest_variance = variance_r;
    i = 0;

    if (variance_g < lowest_variance)
    {
        lowest_variance = variance_g;
        i = 1;
    }

    if (variance_b < lowest_variance)
    {
        lowest_variance = variance_b;
        i = 2;
    }

    /* split the selected axis into two new boxes */
```

## Listing 3. Splitting a Colorspace Box (Continued from p. 39)

```
    ir1 = ir; ig1 = ig; ib1 = ib;
    jr2 = jr; jg2 = jg; jb2 = jb;

    switch (i)
    {
        case 0:
        {
            jr1 = pick_r + 0; jg1 = jg; jb1 = jb;
            ir2 = pick_r + 1; ig2 = ig; ib2 = ib;
            break;
        }
        case 1:
        {
            jr1 = jr; jg1 = pick_g + 0; jb1 = jb;
            ir2 = ir; ig2 = pick_g + 1; ib2 = ib;
            break;
        }
        case 2:
        {
            jr1 = jr; jg1 = jg; jb1 = pick_b + 0;
            ir2 = ir; ig2 = ig; ib2 = pick_b + 1;
            break;
        }
    }

    /* shrink the new boxes to their minimum possible sizes */

    shrink_box(ir1, ig1, ib1, jr1, jg1, jb1,
               &ir1, &ig1, &ib1, &jr1, &jg1, &jb1);

    shrink_box(ir2, ig2, ib2, jr2, jg2, jb2,
               &ir2, &ig2, &ib2, &jr2, &jg2, &jb2);

    /* update statistics */

    sum(ir1, ig1, ib1, jr1, jg1, jb1,
        &total_weight1, &tt_sum1, &t_ur1, &t_ug1, &t_ub1);

    total_weight2  = total_weight - total_weight1;
    tt_sum2 = tt_sum - tt_sum1;
    t_ur2       = t_ur - t_ur1;
    t_ug2       = t_ug - t_ug1;
    t_ub2       = t_ub - t_ub1;

    /* create the new boxes */

    old_box->variance      = variance(total_weight1, tt_sum1, t_ur1,
t_ug1, t_ub1);
    old_box->total_weight  = total_weight1;
    old_box->tt_sum     = tt_sum1;
    old_box->t_ur       = t_ur1;
    old_box->t_ug       = t_ug1;
    old_box->t_ub       = t_ub1;
    old_box->ir         = ir1;
    old_box->ig         = ig1;
    old_box->ib         = ib1;
    old_box->jr         = jr1;
    old_box->jg         = jg1;
    old_box->jb         = jb1;

    new_box->variance      = variance(total_weight2, tt_sum2, t_ur2,
t_ug2, t_ub2);
    new_box->total_weight  = total_weight2;
    new_box->tt_sum     = tt_sum2;
    new_box->t_ur       = t_ur2;
    new_box->t_ug       = t_ug2;
    new_box->t_ub       = t_ub2;
    new_box->ir         = ir2;
    new_box->ig         = ig2;
    new_box->ib         = ib2;
    new_box->jr         = jr2;
    new_box->jg         = jg2;
    new_box->jb         = jb2;

    /* enter all splittable boxes into the priory queue */

    i = 0;
    if ((jr1 - ir1) + (jg1 - ig1) + (jb1 - ib1)) i = 2;
    if ((jr2 - ir2) + (jg2 - ig2) + (jb2 - ib2)) i++;

    switch (i)
    {
        case 0:
        {
            heap[1] = heap[heap_size];

            heap_size--;

            if (heap_size)
                down_heap();

            break;
        }
        case 1:
        {
            heap[1] = new_box;

            down_heap();

            break;
        }
        case 2:
        {
            down_heap();

            break;
        }
        case 3:
        {
            down_heap();

            insert_heap(new_box);

            break;
        }
    }
}
```

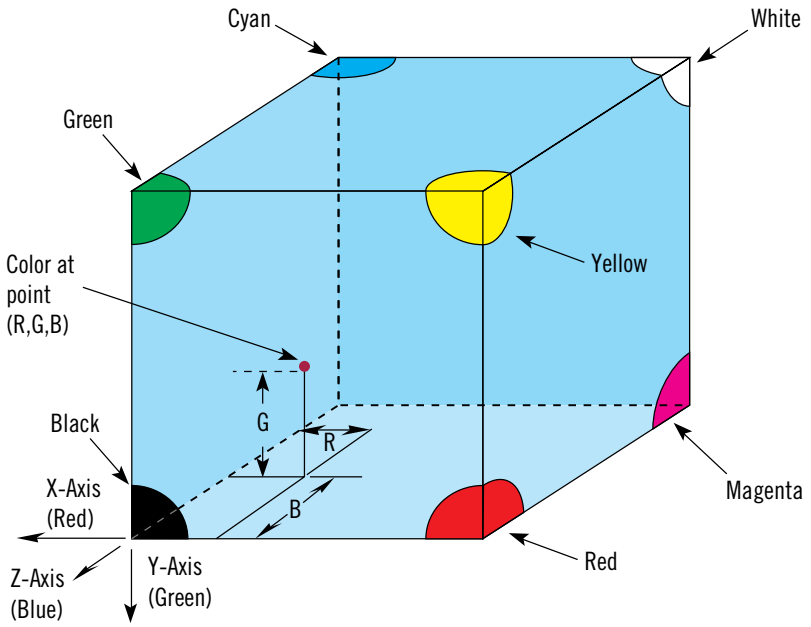
*(Continued from p. 37)*
statistical center of all the points in the box, taking their usage into account.

### A Better Split

I think of Heckbert's statistical and median cut methods as naive. When they split a box into two smaller boxes, their criteria doesn't directly result in reduced color error in the new boxes; sometimes it does a good job, sometimes it does a not so good job. The Heckbert methods either split along the largest axis (making the boxes as square as possible) or distribute the number of used color points evenly, without regard to how close they are to the representative color. Other variations on the splitting criteria exist, but they do not improve the overall result.

In variance minimization, the goal

## Figure 1. The RGB Colorspace Cube




is to split the boxes in such a way that if you added up the variance of both boxes, it would be the smallest number possible. Wu used summed area tables, but we simply used brute force, which turned out to be quite fast and efficient. This function, `split_box`, is shown in Listing 3.

Starting with the box to the split, a plane is passed through the box along one axis and used to divide the box in two. The variance of each box is calculated and the plane's position is to remember if the total variance of both boxes is the smallest yet encountered. As the plane makes its way through the box, individual points are moved from one box to the other by subtracting and adding their values which simplifies and optimizes the process of keeping the variance calculations up to date. The function we used to get statistics for each box is shown in Listing 4.

Once done, another plane is passed through the box along the two remaining axes, resulting in the evaluation of every possible way to split the box. The box is then split along the plane that resulted in the smallest total variances, and both new boxes are placed in a priority list according to their individual variances.

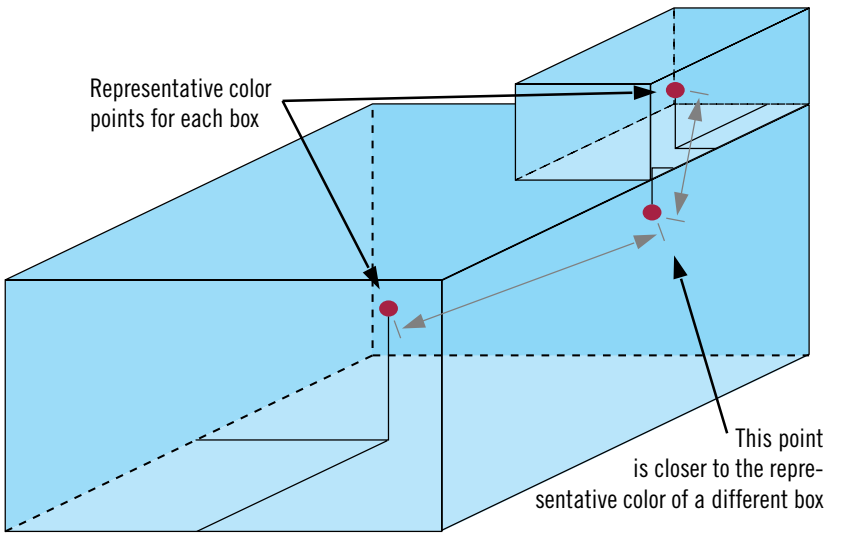
Now, we must choose another box to split. This is where another important difference occurs with variance minimization. Heckbert's method will select the next box based on which box is largest, which box has the most pixels in it, or some combination of the two. In variance minimization, the box with the highest variance, that is, the one that needs the most work to reduce quantization error, is chosen next, and in our implementation it just happens to be the box at the top of the priority list. The process is repeated until the number of boxes created equals the number of colors desired.

An interesting side effect of this algorithm is that you can easily compute how well the image has been quantized at any time by summing up the variance of each box created so far. An interesting variation would be to keep splitting boxes until the total variance falls below a set threshold. This would let you compute how many colors are needed to quantize an image and get a certain quality level.

### The Inverse Color Map

Once we have finished splitting cubes and determining representative colors, the new palette is built by putting each box's representative color into a list. To get from the original 24-bit color to the proper palette entry, a structure commonly known as an inverse color map is created. This is a three-dimensional cube the size of the histogram where each point in the cube contains the palette entry number for the color at those coordinates to be transformed into. In many implementations, the inverse color map is built by filling in each box's area with

## Figure 2. Filling in the Histogram Boxes



that box's representative color number. That is *not* the way to do it. (I pull my hair out every time I see that.)

Consider the situation shown in Figure 2. You have a large box and a small box, which are touching each other. Both have representative colors located approximately in the center of the boxes. In the large box, a point exists right next to the edge where the small box is. Even though it is in the large box, it is much closer in the color space to the small box's representative color. By just filling in the boxes, the pixels of that color will not be translated into the best available color.

There are ways to build the inverse color map so that each coordinate is mapped to the nearest representative color, regardless of what box it is in. We have chosen the method published by Spencer W. Thomas ("Efficient Inverse Color Map Computation," *Graphics Gems II*, Academic Press Inc., 1991). His code is included in the example program available on the *Game Developer* ftp site.

## Performance

In terms of color quality, what is seen by the human eye matters the most. What appears to be a strength of this algorithm is the results it produces when quantizing to small number of colors. For statistical results, Xiaolin Wu reported that the mean quantization error of variance minimization was one-third to one-ninth that of the older algorithms. Our own experiments yield similar results. The difference in results gets greater as the number of colors gets smaller. For a game where only a portion of the palette can be spared for a graphic, this is good news indeed.

What about speed? In his paper, Xiaolin Wu was able to quantize a 256-by-256 image in 10 seconds on a Sun workstation, using five bits of color. In our PowerView program, we can quantize an 800-by-600 Targa file in about one second on a 486/66 in real mode. The results get even better in protected mode, where it is practical to use six bits of color.

## The Final Curtain

Image processing is a complex subject, and we had to leave out some issues to avoid turning this into a book. It's pretty clear that a large and growing amount of game graphics are coming from video, scans and other true color sources. With that, there is no reason not to use the very best methods available to get them into games and other programs, as long as they are practical methods. We hope that by presenting a programmers-eye view, and some practical working code we'll speed that process up and maybe even help some of next year's games look a little more vivid and stunning. Until next time, happy hacking. ■

*Matt Pritchard and Rich Geldreich Jr. work together at Innovatix, a small software company in Garland, Texas. You can contact them via e-mail at matthewp@netcom.com, inovatix@netcom.com, or through* Game Developer *magazine.*

### Listing 4. Gathering Statistics for a Colorspace Box

```
/*---------------------------------------------------------------------------*/
/* Calculate statistics over the specified box.                              */
/*---------------------------------------------------------------------------*/
static void sum(int ir, int ig, int ib,
                int jr, int jg, int jb,
                uint *total_weight,
                uint *tt_sum, uint *t_ur, uint *t_ug, uint *t_ub)
{
    int i, j, r, g, b;
    uint rs, ts;
    uint w, tr, tg, tb;
    uint *rp, *gp, *bp;

    j = 0;

    tr = tg = tb = i = 0;

    rp = hist + ((ir * R_STRIDE) + (ig * G_STRIDE) + ib);

    for (r = ir; r <= jr; r++)
    {
        rs = r * r;
        gp = rp;

        for (g = ig; g <= jg; g++)
        {
            ts = rs + (g * g);
            bp = gp;

        for (b = ib; b <= jb; b++)
            if (*bp++)  /* was this cell used at all? */
            {
                w   = *(bp - 1);
                j  += w;
                tr += r * w;
                tg += g * w;
                tb += b * w;
                i  += (ts + b * b) * w;
            }

        gp += G_STRIDE;
        }

        rp += R_STRIDE;
    }

    *total_weight   = j;
    *tt_sum            = i;
    *t_ur          = tr;
    *t_ug          = tg;
    *t_ub          = tb;
}
```

# The Top 10 Design Sins

*An expert is someone who knows not only the right way to solve a problem but lots of wrong ways.*
*—Marvin Minsky*

To learn how not to make a game, play at least part of a new one every day. Every week, play one all the way through. I did that for more than 15 months for Sega of America's third-party licensing department, part of whose charter is to play, evaluate, beat, and, if possible, crash every game from every publisher for every single Sega platform. And when a game was technically sound, but experientially deficient (that is, not fun), we had to have at least 10 suggestions for improvement. As they say, someone had to do it.

We saw the same mistakes being made across game categories and platforms (yes, we looked at titles for Super NES, 3DO, Jaguar, Duo, PlayStation, PC, and Macintosh, too) by publishers and developers. Each title could have been a better game, often without requiring any more memory; usually it would have cost only another week or two of design time. Following are the top 10 design sins—ones my colleagues and I saw over and over again—that made us shake our heads and ask, "What were they thinking?"


## 1. Boring Levels

A dumbfoundingly oft-forgotten ingredient of game design is the very world a player steps into. Its particulars are frequently assigned to the most junior designers. "Interactive scripting" boils down to "level layout." For that matter, both are direct descendants of the "mission and scenario design" that has been done for boardgames since the 70s, when Avalon Hill and Simulations Publications Inc. (SPI) popularized historically accurate—yet playable—war games.

- The same enemies occur over and over. There are no surprises.
- Games are too linear and provide only one way through.
- Progress through the game does not teach or require a growing repertoire of skills.

Designer Paul O'Connor, of Alexandria Inc., advances a push and pull theory of level design, where players are simultaneously pushed by enemies and pulled by the allure of power-ups throughout a level.

An action game without good levels is just a bad action game. However, good levels without action still make for a good puzzle game.

## 2. The Sin of Repetition

In theater they say there is only one unpardonable sin, "Thou shalt not bore." Level design is only one way to commit this offense.

For action games:

- Background graphics and obstacles are recycled from level to level with little or no change.
- The same attacks work over and over. No new weapons, or abilities, or storyline advances occur as player advance through levels. Players do not get (or are not required) to mix up their behavior patterns.
- Mid-level "continue" markers are scarce or missing altogether, imposing too much time catching up to the difficult parts after every "death."

For strategy and role-playing games:

- Menu structures are too deep. Players find themselves playing with the cursor more than the game. This most often strikes video game ports of computer games, when insufficient thought (or none at all) has gone into reworking the user interface for a joypad instead of a mouse.
- The same maintenance tasks must be repeated incessantly. In 1978, a board game designer named Jim "Bear" Peters coined the term "grocery game." It applies all too well today to any game that forces players into repeated back-and-forth tasks.

Designers should let the computer do the drudgery; the player's input should be the spark that drives the enterprise.

### 3. Not Entertaining (Enough)

Not boring is not enough. Sid Meier of MicroProse put it this way, "Who has the most fun in a game: the player, the computer, or the designer?" The designer did if the game is tricky in a cheap way. The computer does if too much happens offscreen.

- The game does nothing new or unseen in other games already or fails otherwise to stand out in its genre. Known as the shovelware syndrome, this situation finds players asking, "Is this really any better than the 16-bit version?"
- The range of player actions is too limited. Action games must deliver action. If you can do only one thing, it's called a "reaction game," and that genre's time is long past.
- "When you win, you still don't win." These games have cheese screens (victory screens that appear when you've won) that are disappointing or missing. Often, there is insufficient storyline reinforcement of success.

### 4. Production Value (Or Flat Bells And Whistles)

A pet peeve among professional game analysts is CD-game music that could as easily have come from a cartridge. Another gripe is presentational sequences that clearly outweigh the rest of the game in budget and production effort (which only conjures images of The Emporer's New Clothes). You do not want people to say, "Great explosion graphics, how come there's no sound?" or "Where's the music?" Players don't care if your budget ran out before you could address these problems.

- Sound effects are sampled at too low a rate.
- Sound is repetitive and annoying.

Jim Cooper

You dreamed last night you got on the boat to heaven, and by some chance had your PC by your side... To achieve the sanctity of a perfect game, avoid these sinful situations.

Music loops are too short or the same voice bites are cued over and over.
- Video sequences are offensive, usually because nonprofessional actors are used, for example, members of the development team.

### 5. Conflicting Aspects of Production

Surprisingly, uniformity is more important than level of quality. No game is expected to depict reality (yet); all a player asks is not to be jarred out of a willing suspension of disbelief by incongruities.
- Music is inappropriate to the mood, action, or audience.
- Artwork does not match the target audience.
- Sound effects do not match the graphic quality or standard level of quality in the game's genre.

### 6. Licenses Left Unfulfilled

Hits-driven or not, our industry increasingly relies on licenses—leveraging proven or expected consumer entertainment utility such as sports, popular books and movies, cartoons and comic books, even packaged goods like brand-name soft drinks. Making a sports game may seem like a fairly intuitive proposition; turning movies into games takes far greater care. It is not enough to punctuate action sequences with outtakes from the licensed product. On the other hand, don't assume that because comics and cartoons lend themselves so easily to video game translation the work is already largely done. It can be an advantage to have to face the challenge of making a simple inanimate object interactive.
- Signature voices and phrases are not included.
- The design does not explore elements of its license's appeal.
- The game could just as easily have had a totally different title or main character.

This final point is the acid test of any licensed game, and one that every licenser should answer for itself before authorizing any licensee's release.

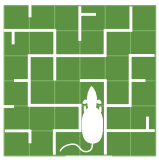

### 7. Counter-Intuitive Weapons and Special Moves

These errors exemplify the difference between showing and telling, between a gloss on design and the kind of gut gameplay players dream about.

Making a sports game may seem like a fairly intuitive proposition. Turning movies into games takes far greater care.

- Powerful attacks are too similar in effect or gameplay.
- Such powerful attacks are too rare or common or too hard or easy to execute.
- Special moves or weapons are illogical in their comparative effectiveness.

Calling a magic spell a "Mega-Bloodcurdling-Turbo-Lightning-Sucking-Farfak-Zapster" only goes so far. The spectacle onscreen and aurally when it's loosed must be impressive and uniquely appropriate. The joypad or mouse button combination to unleash it ought to be demanding, yet intuitive. The effect in game terms should be devastating, but balanced by design elements to preserve it from trivialization, such as scarcity of vital ingredients or limited circumstances during which it is possible.



### 8. Difficulty

Properly setting a game's challenge is a thorny challenge itself. If a game is too easy overall, it's boring. But so is infinite incrementing of difficulty; nobody plays a computerized game as a duel to beat a machine. That's nineteenth-century (and early 1980s) thinking.
- Optimal strategies exist and are obvious. The AI is weak or does not respond to player actions at all.
- Ramping is uneven or flat. The flow of difficulty is uneven—it is too easy and then too hard, too hard and then disappointingly easy, or never changes at all.
- Bosses—arch-villains to provide final challenges at the ends of levels—are missing, too easy, or too hard. Only one or two tactics are needed against bosses. Enemies or bosses don't take damage like the player does.

Beware of creating pseudodifficulty, however, such as "mandatory hits," or damage that cannot be escaped at any skill level, even if nonfatal. Trying to lull the player into a false sense of insecurity is a no-no.

### 9. Execution

A major game design sin is allowing the challenge to arise more from navigating a game's interface than negotiating its story elements. Programmers and designers are most prone to this mistake; they cannot help but think learning and doing things the hard way is fun. It is not. It's work, the kind they are paid to perform for the players.
- Collision detection is too loose or

tight. For example, when collision detection is too loose, players experience fighting game "hits" that should or shouldn't have landed. When it is too tight, players must position a climbing or jumping character's sprite at precisely the correct pixel.

- Animations of actions are poorly timed or have insufficient frames. Games simply lack state-of-the-art graphics and sound.
- Basic character movement physics are lax. These physics include jumping range (which might be too far or too short), hang time (anything outside a range between apparent earth standard and moon gravity needs justification), speed of motion, and acceler-o-meter functionality (after a half-second moving in the same direction, a character or onscreen mouse arrow should speed up.).

If you aren't able to create the next new technology, you should at least be able to execute the current standard well.

### 10. Low Replay Value

Games differ from books and movies in the number of times they are consumed. Games with linear levels give players no reason (or opportunity) to revisit scenes from new angles. The investment in the average game's technical development could stand to be exploited even more. And no matter what your graphics and sound budget was, after players see and hear those jewels a few times, the only thing they want to wrestle with is what you worked on for just two months out of the development cycle—the game.

- Variable difficulty settings are missing.
- High scores or low times are not recorded, robbing players of the challenge to try and beat them.
- Players don't have a selection of playable characters to choose from.
- Insufficient "player-override" functionality. Players cannot skip cinematic scenes and special effects.

This last is a cardinal sin all its own; by unnecessarily fatiguing players, it mortally wounds replay value. Always, always remember the experience you offer will be consumed iteratively, that is, over and over again. Make maximum user configurability your friend.

## GO, AND SIN NO MORE

**Everyone participating in the design process can help avoid the 10 sins of game design. The burden should not fall on the developer alone.**

**Designer**

- **You are not just designing a game, you're crafting game elements to survive the maelstrom of producer, marketer, and licenser reviews. Stick to your guns and learn to creatively outfox upper management. It may seem like a game unto itself, but you must play for you and your game to succeed.**

**Producer**

- **Do your job. It's not just spreadsheeting, or phone-mongering, or lunching. It's overseeing and delivering the game from the designer to the distribution channel.**

**Marketer**

- **A high-variance strategy is actually your best career bet when dealing with games. At early concept meetings, champion the designer. If it pans out, you can say you pushed for it. If it doesn't, you tried to talk market sense into them. When the game is almost ready to launch, you must look beyond feature lists at tactical marketing meetings. It's your job to figure out how to bring the game alive. Have the producer explain what's unique about the game. It will take more than a few screen shots to communicate that to potential players.**

**CFO**

- **Quality sells. Much has been made of the unique marketing behind Doom and Descent. Don't forget—there was unprecedented quality in each game. So why not put that incremental product development money your producer is begging for into next quarter's marketing budget, under word-of-mouth advertising expense?**

**President**

- **To get the best out of your people, give them the best of yourself. Are you duty-bound to maximizing return on investment or increasing shareholder value? Your franchise as a publisher depends only on a reputation for quality product. Believe it or not, Spectrum HoloByte's stock rose the last time it announced another delay in getting out its first Star Trek: The Next Generation CD-ROM game. Perhaps this unexpected reaction from the "smart money" had something to do with the lesson the entire industry learned when another publisher's bug-riddled graphical adventure CD, The Lion King, experienced retail return rates rumored to have exceeded 50% of sales.**

## Dos for each Don't

Fortunately, where I come from, you don't complain about problems without offering solutions. Here are a number of ways to avoid the 10 sins I've outlined:

### 1. Boring Levels

- Make demands on players. You must know their capabilities, which is admittedly not always easy.

### 2. Repetitiveness

- Don't stint on the preparation of a great storyline. Treat the game more as a novel than a short story repeated over and over—or as a movie, not a music video. If everyone thinks he or she can make a video game, show why they are wrong.

### 3. Entertainment

- Give good value.
- Dare.
- Lead, as opposed to manage.

### 4. Production Value

- Have a production concept that somehow fits on the back of your

business card and still serves as a yardstick against which every development decision can be measured.

### 5. Production Coherence

- Manage the development process, that is, its participants. All are necessary, none is sufficient alone. When appropriate, tell them so.

### 6. Licensing

- Build a game from the license up; don't just plug and ship your game (or let your licensees do it for you).
- Know, define, and insist on making central to gameplay the specific appeal of your licensed property. If the license has any value at all, playing to its strengths is a recipe for success.

### 7. Special Moves

- Extend the metaphors in your game. You cannot tell players they are having fun. They have fun by being fully engaged. They will only be fully engaged if they acknowledge and accommodate differences you define. Make the player's input and the resulting game outputs consequential while internally consistent or the whole house of cards comes tumbling down.
- The soul of video gaming is the epiphany of acts of desperation that become intention. In a fighting game, it's that joystick-slap or button-slam combination—born of a split second's agonized frustration—that miraculously looses the perfect mid-air counter-throw. Like math, it makes "of course" sense after you've done it.

### 8. Difficulty

- Defy passivity at every turn.
- Surprise, but play fair.

### 9. Execution

- Optimize the inevitable compromises feature for feature by weighing benefit against benefit in play value. Use marketing for focus groups and user testing throughout the project.

### 10. Replay Value

- Roll at least three games (if not five, or 17) into one. *Star Trek* and *Star Wars* appealed to children, teens, and adults through spectacle, action, and ethical components. Three types of people could enjoy them, and anyone can still enjoy them on three or more levels.

ACKNOWLEDGEMENTS

**Grateful acknowledgment is made to Steve Ackrich, group director, Sega of America Third Party Licensing and Acquisitions, for permission to print this article based on materials I prepared and delivered with him at the Sega Developers Conference, March 1995. I'd also like to thank him for admission to gameplay bootcamp. And to Gary Barth, my drill instructor, now at Sony Computer Entertainment of America and the producer of Battle Arena Toshinden.**

### The Eleventh Sin

The 11th sin is failing to realize these are not design sins, but management sins. No competent game designers willingly commit them—their masters do. As Gordon Walton says, "Everything in this business conspires against good games."

The platform wars among 32-bit consoles have already spread to the PC. It happened when Windows 95 began promising true plug-and-play ease of use. Race your products to market, but avoid these simple errors. It's worth it to take an extra three weeks design time up front and tack another two to three weeks onto the end of the cycle for polishing.

Marketing myopia is the technical term for the flip side of the "not invented here" coin. So you have the technology. Your game can do what has never been done before or something wildly better than has ever been done. That's not why people buy it. They will buy it—and continue buying it—because it's fun. ■

*Jim Cooper, a gamer with an MBA, has founded software and game companies, consulted to money managers, and taught marketing. He is the founding editor of* The CGDA Report, *and can be reached at 72147.2102@compuserve.com.*

# The Play's The Thing


**Hounds and Jacyls: Integrating puzzles into a story adventure without disrupting the player's suspension of disbelief is one of the biggest challenges of interactive stories. In Hyperquest's new title, Archeologica, the game puzzles not only move the story forward , they give the player a deeper connection to a central character in the game.**

How do you keep a game interactive and also weave in a good story? This question is becoming more important in the gaming world. It's possible now to make stories somewhat variable and interactive and to make great game experiences more dramatic and character oriented. But the possibility of truly merging the interactivity of games and the suspension of disbelief of drama is the carrot dangling before many multimedia professionals today. Yet, is this merger possible? "It seems like the term 'interactive story' is an oxymoron," says Jonathan Knight, a producer with Viacom. "It seems like you can't have one and the other. It's one or the other."

Still, the possibility is too cool to ignore, and so game developers continue to search for ways to marry the two art forms. They've come up with many approaches, some more successful than others. "We don't have the polish of the thousands of years of work that has gone into regular stories," says Lawrence Schick, managing director of the interactive story and strategy group at Magnet Interactive. "In comparison to what a playwright or screenwriter does, the stuff we do is awkward and unpolished, and it isn't as satisfying."

People working with interactive storytelling in any form it may take—be it within the confines of a shoot-em-up action game or a puzzle-oriented mystery game—face many challenges as they search for the key to a successful interactive experience. Here are a few things they're wrestling with.

## The Illusion of Player's Choice

Many interactive story games have the ultimate goal of giving the player the feeling that his or her decisions are actually affecting the story. "That's the holy grail," says Schick. "Obviously, within very strict limitations, you want the player to write the story by the decisions they make." This has been a tough goal to reach, so far. Branching plot stories, in which a story unfolds as the players make various decisions, are one approach to interactive stories, but the story must be limited or the decision

tree becomes astronomical. Graphic adventures, where a story is revealed to the players as they succeed in solving puzzles or winning battles, are another approach. This genre, which includes the Kings Quest Series and Wing Commander III, offers gameplay and story together, but you have story segments followed by gameplay segments. The two aren't integrated.

Computer role-playing games based on building up skill levels and possessions, such as Sword of the Samurai, let you model a limited set of a character's personality traits—abstracts such as a character's aggressiveness, hostility toward the player, and honor—and manipulate the numbers with algorithms and routines that give the impression the characters are reacting to events in the game.

You can also incorporate a dramatically correct story into the process by submitting plot elements to the same quantification. You can set up algorithms that weight a character's options to mirror another character's (so it appears that one character is following another), hold back options for dramatic effect (only after your player's character marries the villain's love can the villain have the option to send assassins after you). This presents the illusion of an unfolding story based on the actions the player takes—the characters appear to react to what the player does. But you usually have a limited number of generic and repetitive dramatic pieces to work with. "The problem is creating something that apparently stitches itself into a movie, without repeating, and without having to be 10 CD-ROMS." says Schick.

## Player as Hero

Viacom's Jonathan Knight looks at the "holy grail" a little differently. To him, the key to a successful interactive story game is aligning the player with the hero's objective. It's all about making sure the player and the hero want the same thing. "That's the way modern drama has always worked," explains Knight. "Stanislavsky felt that every story ever written hinges on the objective of the hero. Whatever the hero wants out of the story will drive that story to its conclusion."

It's this element that Knight feels makes highly interactive games like Asteroids and Doom, and even more puzzle-oriented games like Myst, so successful. "Every game back to Asteroids is really an interactive story," says Knight. "In Asteroids you have a hero (the little ship), you have antagonists (the rocks), there's setting (space), you have rising action (more rocks coming at you), and there's climax—it always ends tragically with the ship getting blown up." The objective is clear to the player immediately, and there's no conflict between what the player wants and what the hero wants.

But Knight says this element is lost in many interactive stories. Sure, it's easy to align yourself with a ship in danger—that's based on "sweaty palm" emotions, fear and survival instinct. But what about objectives that are more complex: something like Hamlet's dilemma, for example, in which our hero's objective is to avenge his father's death. We might be drawn to this objective deeply enough to observe passively at a Shakespeare festival, but are

Barbara Hanscome

**All the rock 'em-sock 'em graphics in the world won't make up for a bland story. Today's designers are rising to the challenge and creating meaningful plot lines as well as beautiful visuals.**

we involved enough to play "Hamlet Interactive?" Plus, the story is already written, how can we really affect the outcome?

Games that are structured so that the core objective is achieved through a variety of sub-objectives work well to keep the player moving along on the right path.

For example, if you were creating the "The Fugitive Interactive," the hero, Richard Kimball's, main objective is to prove his innocence in the murder of his wife. But in Act 1, his mini-objective is to escape incarceration. In the film, he achieves this objective by jumping out of a bus, disguising himself as a doctor, stealing an ambulance, hiding in a sewer system, and jumping off the top of a dam. "We can offer five more, or 10 more, to the player, so that they're not forced down the path the film follows," explains Knight. "These different actions could suit a whole range of personalities, and yet, sort of unknowingly to the player, the character is still achieving the objective that we have set for him.

But that doesn't completely solve the problem of merging the hero's objective with the player's objective. There will always be options the player will want to take that aren't options presented in the game. What Knight suspects might be the key is simple, Pavlovian condition/response. "Because stories are so psychologically complex, and the distance between what the hero and the players want is so great, I think we need to use animal conditioning on our players, and basically reward and punish them psychologically, right in line with the objectives of the story for certain behaviors." He admits it sounds kind of diabolical, but it happens all the time in games. "If you think back to Asteroids, if you didn't destroy the rocks like you were supposed to, if you just sat there and cruised around and didn't go after the objective, then they started playing this music. It makes you really nervous and you get really scared. And if you go after the rocks, it stops. They're conditioning you."

But, how does Pavlovian conditioning fit into a complex story with characters, dialogue, and plot? "If you're really going to condition someone, you can't be too obvious about it. You don't want to kill them if they walk though the wrong door, for example. That's not conditioning, that's just irritating," says Knight. The goal is to reach the player at a deeper level. "You don't want to reward and punish actions as much as want to reward and punish emotional responses. Emotion is what is deep down and subtle, and that's what the player is not going to be conscious of."

## Making Feedback Work for You

Knight admits he's still working on exactly how to create this subtle emotional conditioning. He thinks feedback from supporting characters might be one solution. "Other characters can work to goad the hero in certain ways. In the Interactive Fugitive, for example, where the goal of the hero is to prove his innocence in the murder of his wife, if a guy came up to Richard Kimball and said, 'You're a murderer. You killed your wife,' and then the player decides Richard Kimball can beat the crap out of the guy, that behavior—even though it's violent—is conducive to the objective. And we want to reward the player for that."

Feedback is also crucial to tell the player that his or her decision made a difference to the story. "Feedback is almost the hardest thing to do," says Schick. "Letting the player know what difference he or she made in a story implies a couple of things. First it implies tipping your hand as to what would happen if the player didn't make a decision, and second, it implies that you have to somehow, in some brutal way, tell the player what difference this made to the other characters, which is what matters in the story. It can feel contrived, but you've got to have it. In an interactive story the players are making the decisions and you have to set things up carefully so that they see the implications and results of their own decisions."

Schick points to the game Johnny Mnemonic as an example of a game that successfully melds action and story, however the feedback is missing. The game is a linear movie in full-screen MPEG until the player has the option to interact, then the screen moves to letterbox. "In Johnny Mnemonic you get to make several kinds of decisions. The hard part is knowing what kind of decision you get to make at any given moment. They don't give you a cue as to what kind of choice you're making: Is it a movement choice, should I go left or right? Should I pick something up or put it down? Is it a fighting option where I should kick and punch or block?" explains Schick. The players have a limited time before the movie goes back to full screen. "When you make your decision matters as much as what you decide to do." says Schick. "If you don't do anything, it often goes into a bad end, you didn't act fast enough and the bad guys blow you away."

Feedback is also necessary to avoid one of the most basic trouble spots of interactive stories: bottlenecks, in which a player cannot move forward until he or she solves a puzzle. Bottlenecks destroy suspension of disbelief and are just plain frustrating for the player. In his games Castle of Dr. Brain and Quest for Glory IV, David Cole provides a help mechanism to avoid this problem. "I tell the player up front that if you click on this button five times, the first four clicks will give you a hint, and the fifth click will solve the puzzle for you." To keep it challenging for the player, the fifth click doesn't reveal the solution to the puzzle, but it moves the player forward.

"They did the same thing in the 7th Guest," explains Cole. "In each game puzzle you can go down to the library and there's a hint book that tells you a little bit about each game puzzle. And if you consult the book several times in a row, eventually it says, 'oh, okay, you solved the thing.'"

But providing help isn't good enough, you have to let the player know help is available. "In 7th Guest, I didn't know that you could get past the puz-

zles by using a hint book until I did it by accident," says Cole. "It's a tricky task, because on one level you want the player to be totally immersed in the fantasy of the game and not thinking about the computer. On the other hand, if you've got these helpful things built in, it's nice to make that obvious to the player that those things are there."

## Integration of Puzzles and Story

Games such as Wing Commander III and Daedelus Encounter come close to merging the excitement of game play with strong characters and storylines. However, it's not really a mix. The story happens, then the fighting happens, and when you're not fighting you're stuck watching a very set script that you cannot control. Integrating puzzles and story is even more challenging for edutainment designers, who rely on puzzles to teach subject matter and test knowledge. How do you make these a seamless part of the experience?

Game producer Julia Mair and her team from HyperQuest wrestled with this problem in their two educational titles, Astronomica and Archeologica, which not only mix gameplay and story, but education as well. In Astronomica, the player learns about astronomy by helping a teenage heroine named Sara save her dad from captivity in an observatory. The player navigates through a photorealistic three-dimensional environment, similar to Myst, and must help Sara reset each exhibit in the observatory to get the computer system up and running. Each exhibit involves an educational game or puzzle to solve. Mair felt the game was successful for many reasons, but the puzzle and story mix didn't hit the mark. "We felt that what was happening was that we were forcing the player to stop the action. We wanted the player to move through the dramatic structure and never, or very infrequently suspend their disbelief to play the games. Many of the games are challenging and fun and interesting, but you still have to stop, you have to play the game, you have to get it, and then you move on."

In Archeologica, the learning aspect is still dependent on puzzles and games, but Mair and her team think they integrated them more successfully. They set up a main objective for the player similar to the one that exists in Astronomica: the player must help two characters, Sara and Elija, find and rescue a character trapped within an environment. In this case, the character is Elija's mother, an archeologist trapped somewhere within an archeological dig site. The games and puzzles are layered within the experience. For example, the players don't just passively hear about the Egyptian Book of the Dead after pressing a button, they also travel through spaces in the dig that represent passages of the book itself. They don't just learn the significance of the Solar Boat to the Egyptians, they maneuver one through a secret passageway. Some of the games are the same games Egyptian children played, and certain plot twists are revealed to the player only by using knowledge they've accumulated in the game, such as deciphering hieroglyphs or using tools archeologists use.

Then they took it one step further. "We didn't want the player to feel that he or she was pitted against how clever the developer can be or grounded in a twitch arcade element only," says Mair. To avoid this, they created another character, an archeologist named Burton who once inhabited the dig site

## TOOL BOX

**Combining an interactive story with a challenging objective requires vision—literally. Here are a couple of tools that can help you visualize the twists and turns of your game's structure from plot line to character development. Use these tools to diagram, outline, and brainstorm projects, and you will begin to understand the art of telling an interactive story.**

**StoryVision, by the company of the same name, is an organizational tool for interactive gaming used to visually diagram plot structure and characterization. Analogous to a detailed blueprint of an interactive script, this tool allows you to map out the "game plan" through the use of bubbles and lines, similar to a flow chart format. Once the scene bubbles are created, you can attach text to the bubbles in any word format. CD Noir's Dave Dengler, who has used StoryVision says, "It's also great for brainstorming." Used by writers, producers, and developers, this program links with any word processing program and can organize the text in a screenplay style. Although the program does not include its own text editor, user Chris Taylor of Interplay Productions says, "I like the program's simplicity. It is quick, easy to use, and makes changes rapidly." For more information contact StoryVision at (310) 392-5090.**

**Inspiration, a diagramming and outlining tool by Inspiration Software Inc., can be used as a brainstorming tool and a diagramming tool. With a choice of over 500 symbols including the full ANSI flow charting symbol set, and color, shape, and size capabilities, this tool makes it easy for you to differentiate between visually linked ideas. If you still haven't found the perfect symbol for your idea, Inspiration lets you import your own graphics. The diagram view also lets you create smaller diagrams that connect to parent diagrams. Lynn Pierson, product specialist, says, "There are hundreds of levels of diagrams. Each diagram has the capability of connecting to a child diagram." Each symbol in the diagram has a linked Notes Window allowing unlimited pages of text, and the multiple zoom feature offers a variety of perspectives to help map out your game's intricate structure. Toggling from the diagram to the outline allows you to view your plot structure or characterization in a text format. Complete with its own text editor, Inspiration can revise, edit, and format in the outline view. For more information contact call Inspiration Software Inc. at (503) 245-9011.**

**—Deborah Sommers**

where the game is set. He has set up the puzzles and traps to protect the treasures he's discovered throughout his life. The player discovers various aspects of Burton's personality through the intricate, Rube Goldberg-like games and puzzles he has created, and falls into a sort of love-hate relationship with him. Instead of interrupting the story with a puzzle, and frustrating the player with a game created by someone else, the puzzles take the player deeper into another layer of the story.

Of course, veterans of role-playing games, mystery games, and adventure games are old hands at this. This is what their games are all about. "There's a misconception in the adventure game field that the puzzles are arbitrarily pasted in. A lot of our puzzles are about relationships," explains Corey Cole. "Figuring out how to interact with different people, how to interact with the bartender, how to meet someone, how to get them to join your party, basically how to help them so they will help you. That makes the game world feel a lot more real. Look at movies, they're all about character relationships, and that hasn't been done very much in games thus far. We're trying to bring much more of that in."

## Living, Breathing Characters

But how do you make those characters interesting enough for your player to want to interact with them? How do you telegraph a character's personality and motives and communicate these traits clearly and quickly to the player? HyperQuest's Mair says it comes down to trading off when stereotypes will work for you and when they will not.

In Astronomica, Mair and her team really wanted to create a strong female heroine, but one that was also a regular kid. "In the first scene, we wanted it to be clear to the player that Sara was a leader, we wanted to show a young girl with a problem who is strong enough to tackle it." In the first scene, Sara makes a decision that will guide the entire game (she must reset each exhibit in the laboratory to bring the computer system back online). This is the main objective that drives the game, so the player needs to trust that her decision is right. The HyperQuest team worked hard on the finer details of the character, choreographing her movements, her tone of voice, and her physical posture so that her personality and situation would come across loud and clear.

When it came to supporting characters, such as Sara's father, an eccentric scientist with less time on screen, they let stereotypes work to their advantage. "We wanted to telegraph his passion, that he's committed to what he does, and that he's more than capable of putting his shoes in the refrigerator. We tried to choose where stereotyping worked for us, because we wanted to cue into a whole set of basic cultural IDs."

> Astronomica and Archeologica, two titles from HyperQuest, not only mix gameplay and story, but education as well.

In games and film you can still leave out a lot of information because the player will fill it in. "It doesn't take much to create a relationship between the player and a character in a story," says Lori Cole, who wrote the Quest for Glory series with husband Corey Cole. "The player will put the imagination into the character if the game has a base for them to build upon." Her favorite anecdote to support this theory has to do with the game Wing Commander I. When players got to the scene in the game where one of the supporting characters died, BBSes were filled with posts from people lamenting the character's death. "People posted things like 'She was my love! We had a relationship!' They really didn't have a lot of interaction with this character, but that didn't matter."

Chris Thompson, an interactive script writer agrees. In his article "Interactive Drama, the Rules of Storytelling" (*Morph's Outpost*, May 1995), he writes, "Developers shouldn't be afraid to withhold information. If a user is interested in a character or situation, there's almost no limit to the amount of 'unknowns' that he or she will tolerate. In fact, unknowns generate intrigue, suspense and mystery."

## What's Next?

Where is the industry going with interactive storytelling? Some say multimedia is waiting for someone like Stephen Spielberg (or Einstein) to come solve all the problems. Some are waiting for the "film/game genre" to die so they can get back to true game experiences. Some say it's going to depend on motion capture technology and artificial intelligence before we'll be able to realize the true integration of interactivity and story. "We're really wrestling with stuff no one has wrestled with before, and for fairly high stakes," says Schick. "So people get either desperate about it, or they basically want to fall back on what they know will work. The only reason why any of it works at all is because the concept of actually experiencing a story rather than watching it is so compelling that people are engaged, despite the embryonic form of the art." ■

*Barbara Hanscome is managing editor of* Software Development *magazine.*

# Creation of a Fantastic World

David Sieks

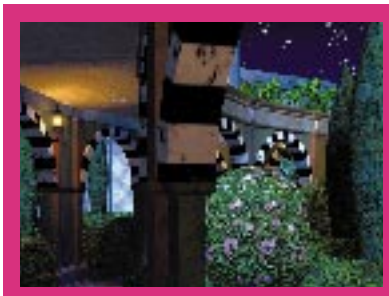
Nothing brings a game to life so much as realistically rendered detail, no matter how minute. This month, David Sieks turns an artist's eye to Buried in Time, the new time travel game from Sanctuary Woods.

Artist's View remains pretty much confined to graphics issues. However, as digital entertainment has matured—has *begun* to mature—it sometimes happens that the borders separating visual considerations from other game elements become blurred. When, you might reasonably ask, are game graphics anything more than just that? The answer: when they are an integral part of the design process of the game as a whole, interwoven from the earliest conceptual considerations with the overall fabric of the game, inseparable from any discussion of the title's creation or its effectiveness. Such is the case with Buried in Time, sequel to the Journeyman Project, labored on for two years by Presto Studios and released this summer by Sanctuary Woods.

Many companies pay lip service to an integrated approach to design, yet, while ever-greater importance is certainly being placed on quality graphical content, in most instances game visuals still occupy a largely decorative role. With

Though a picturesque ruin is all that remains today of Richard the Lionheart's 13th century castle, Presto artists turned to contemporary sources to recreate it in stunning detail for Buried In Time.

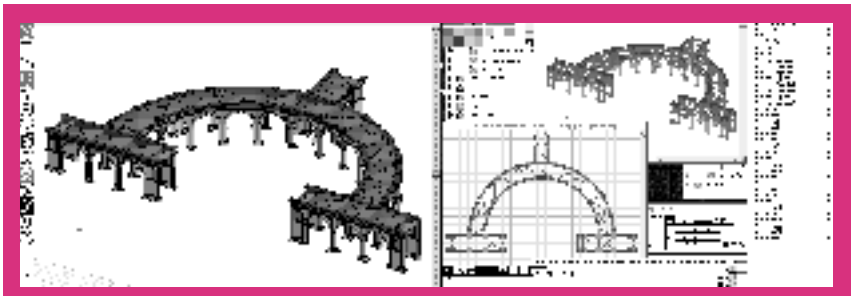
Buried in Time, graphics do not serve as mere adornment, however. Nor for that matter is another important creative element—story—just a weak scarecrow frame on which to hang gameplay. While Presto's ultimate objective was a fun, playable computer game, there was also a desire to create a deeper and richer experience than the typical adventure title. Presto's president and Buried in Time producer Michel Kripilani, writer Dave Flanagan, and creative director Phil Saunders spent more than five months assembling the skeleton of a game to realize that ambition, all before even going into preproduction. From the very start, visuals, gameplay, and story went hand-in-hand, and that integrated approach is apparent in the way the game works so successfully to draw the player into its world—or rather, worlds.



All the painstaking research and careful design is brought to life by artistic use of textures and lighting.

## Setting the Scene

In Buried in Time, the player assumes the role of a time-travel agent wrongly accused of tampering with the past. To vindicate yourself, you must escape house arrest and journey through time and space to the several zones in question in search of clues to the identity of the true culprit (solid adventure game stuff, but still a welcome change from the onus of saving the universe yet again). During the early design process, the team at Presto considered many time-travel destinations. One chief criterion was visual appeal; Saunders wanted periods and places that would lend themselves to the lush three-dimensional graphics he had in mind. Freshness was also a factor (Egyptian pyramids failed that cut), and of course each destination had to offer gameplay potential.



Presto designers did their homework to ensure the several historical settings were accurate and convincing in their detail, even for places that never existed, like this cloistered garden path for a palatial workshop Leonardo DaVinci might have had in Milan— but didn't.

The Presto game design team further complicated their task by opting for historical accuracy, despite the fantastical plot. This meant exhaustive research, not only for visual and descriptive references on which to base designs, but also for convenient gaps in known history that provided opportunity for embellishment. Eventually, along with several futuristic environments, three historical settings were chosen: the medieval castle of Richard the Lionheart, a Renaissance era workshop of Leonardo da Vinci, and a Mayan ziggurat.

The first two proved especially fertile subjects for Saunders and supporting conceptual designer Victor Navone to build upon. Little remains today of Richard's Normandy castle, Chatteau Gaillard, which fell in the early 13th century, but Saunders used contemporary sketches and descriptions to fill in many of the gaps, and was able to extrapolate the rest to recreate a wholly convincing environment. On the other hand, despite its impressive architectural detail, the da Vinci workshop is pure invention, hypothesizing the great man's whereabouts and activities during a shadowy period in his history. Again, due to meticulous attention to known details—and equally meticulous fudging of the rest—the setting attains an eerie degree of authenticity.

Throughout the two years it took to complete Buried in Time, environment design, story, and gameplay elements advanced apace. From the early plot and environment decisions, Saunders moved on to conceptual sketches, Flanagan continued to flesh out the story, and Kripilani began to assemble the production team. Much back and forth was involved, and many changes were made along the way, but the basic framework that had resulted from those first months of brainstorming remained in place to hold the project together through its growth. Though time-consuming, such an integrated approach helped to create a cohesive and balanced game, with a story that carries you along while the detailed environments draw you in.

## Designing for Immersion

The game's graphics feature rendered, point-of-view interaction spots linked by full motion sequences. The level of visual detail in Buried in Time is astonishing and, combined with excellent sound effects and an atmospheric score, contributes to an environment that envelopes the player like a warm bath. Drawing the player deep into the game world was the goal, and no effort was spared to accomplish this end. At times, this was a grueling commitment to stand behind, as so much of the detail is extraneous to actual gameplay. But the Presto philosophy is that detail is what convinces the player to suspend disbelief and enter the illusion, and when it comes to illusion there's no such thing as being too convincing. Its 40 minutely detailed environments contain roughly 10,000 three-dimensional objects. By the end of the project, Presto artists had more than 40GB of visual data online.

One reason all these details achieve such a convincing effect is likely Saunders' background as an industrial designer (he's still one of the lead car designers

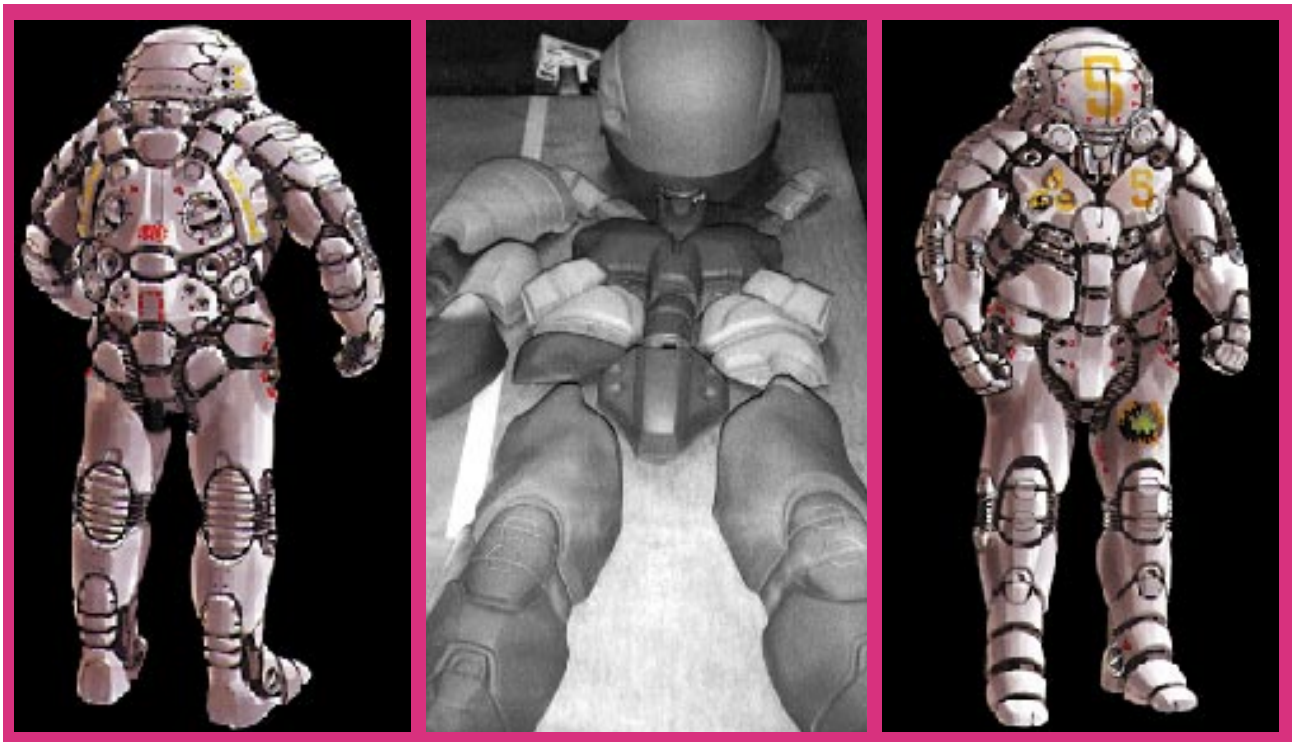
at Nissan). Even fantastic science-fictional settings are grounded by his practical form-follows-function approach. He also has an obvious flair for the dramatic (before automobiles, he designed theme park rides) and these strengths provided the guiding vision for the Presto art team, a vision Saunders communicated by multitudinous sketches, marker comps, and lots of arm waving.

Though their creative director knew what he wanted to see and provided a strong conceptual starting point, the team delegated completion of the artwork to various departments, where the visual guidelines were fleshed out and further details added. Conceptual sketches were turned into three-dimensional models using Form-Z, from Autodessys Inc. Three-dimensional artists Jose Albanil and Leif Einarsson found Form-Z an excellent and very robust program, and as one of the first to offer Boolean operations on the Macintosh, this tool allowed rapid creation of the many complex objects in the game. The three-dimensional models were then passed to E.J. Dixon III and Frank Vitale for texture mapping. Rather than simple nontiling textures, materials and relief maps were created in Photoshop (Adobe) that exactly fit each object.

Finally, the completed three-dimensional environments were lit, animated and rendered with Electric Image (Electric Image Inc.), which Kripilani unreservedly describes as "hands-down the best 3D animation package available for the Macintosh. Nothing else even comes close." An optimized Phong renderer obviated the need for time-consuming ray tracing, allowing animators Shadi Almassizadeh, Eric Hook, and Eric Fernandes to complete renders in a fraction of the time while maintaining a high level of quality.

Though a different department handled each stage of the production, there was much communication between the various artists involved. Often, pieces had to be sent back up the chain when a texture didn't work quite right or an animation required an object to be articulated differently. The guiding dictum was that it wasn't done until it was right. Shepherding it all along was project manager and vice president Farshid Almassizadeh (yes, they're brothers), whose job it was to see that both schedule and high standards were maintained.

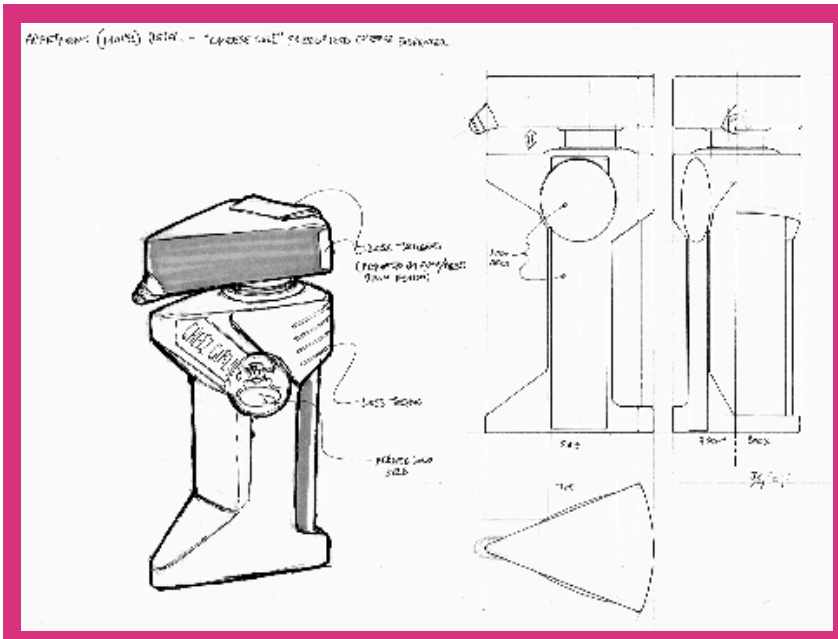
All the graphics for Buried in Time were generated on Macintosh computers, mostly PowerMacs, with PowerMac 8100/100 systems sporting 140 MB RAM and 2GB hard drive space as render servers. As Kripilani explains, the best graphic creation tools are available first for the Macintosh platform. "When you are on the bleeding edge, like we are, you need the best tools as soon as they are available."



**The decision to incorporate live action video rather than graphically rendered computer 'actors' drove the design of the time-travelling biosuit, which was transformed into an actual costume by a Hollywood special effects company.**

## The Well-Dressed Time Traveler

With such photorealistic detail going into the environments, the team decided to use live action for the occasional human presence. Even the best attempt at a virtual actor would not be realistic enough, they felt, and would fracture the illusion Saunders and the Presto artists had worked so hard to create. But incorporating live action presented its own problems and proved a complicated process.



The wealth of detail in Buried In Time called for designs running the gamut from a sprawling space station all the way down to this surprisingly useful handheld cheese food dispenser.

The first step was finding an actual video camera lens that would match the perspective of the rendered environments. This done, Saunders was able to design ahead of time the environments in which the digitized actors would appear. On the set, marks were placed to guide the actors through the virtual terrain, and lights were situated to replicate the lighting effects that would appear in the rendered environment. For example, a yellow light connected to a flicker box created the effect of torchlight for the castle scenes. Composited into the rendered scene (with Aldus/Adobe After Effects), the digitized actor looks more convincingly a part of the picture than previous experience of composited video and computer graphics might have led you to expect.

The decision to use live action generated another issue: creation of the "biosuit," a bulky, armored personal time-machine worn by the hero and his fellow temporal agents. Further, the fact that this item would have to be a real object, worn by a live actor, forced certain design considerations. The original plan had been for a suit composed of large, highly reflective metal plates. First, the large plates had to be segmented to allow movement. Then the reflective surfaces had to be changed to a very matted metal to avoid picking up reflections of the blue screen in front of which the video was to be shot. In its final state, the design for the biosuit looked like a cross between an old-fashioned deep-sea suit and the robot from *Lost In Space*.

The actual suit was constructed by All Effects Company in Hollywood, Calif., using a complicated process in which each section of the suit was carved from foam, following Saunders's design specifications. A mold was then pulled from the foam carving, a plaster cast pulled from the mold, and finally a vacuform plastic piece pulled from the plaster cast. The pieces were sewn onto a foam bodyform to give the suit bulk without undue weight (though the final product still weighed in excess of 40 pounds). Special removable identification markings were used, so that during the game several

Death scenes, painted in loving detail by Gary Glover, accompany a description of the agent's ignominious demise (in this case, lactose intolerance).

agents could be shown suited-up, though only one costume had been built. Before you rush to order yours for next Halloween, you should know that the gang at Presto considered this suit a deal at $25,000. (For that price, you could buy yourself a nice Nissan.)

Besides being a neat costume, the biosuit also provided a logical game-world excuse for the interface. Because you, the time-traveling player, are constantly sealed within this futuristic Michelin Man getup, your view onto the world is through the suit's video display camera, which is fed to a smallish screen in the center of the interface, surrounded by inventory and other controls. These controls are simple, straightforward, and work so well that they quickly become all but invisible to the player, who is by then engrossed in the beautifully rendered scenes in the view window. To the user, this interface is so deceptively simple that if you've never attempted to design one yourself, you might never credit that it was the result of three months of teamwork.

In almost no other regard, however, does Buried in Time appear to be a game that was easy to make. If you can step back from the illusion long enough to admire the depth of detail evident at all levels of this game, you realize that it has gut-busting effort written all over it: "It's amazing how draining these projects can be," says Kripilani. "We are still trying to bring our brains back from jello-land." ■

**Table 1. Buried in Time Statistics (Figures are Approximate)**

| Item | Number |
|---|---|
| Design documentation and script | 300 pages |
| Environments, rooms | 40 |
| Preproduction sketches | 1,000 |
| Individual three-dimensional objects | 10,000 |
| Total number of polygons | 10,000,000 |
| Textures | 6,000 |
| Animation, frames | 30,000 |
| Visual data in final game | 25GB |
| All-nighters | 100 |

**The settings in Buried in Time are so rich and compelling that you are prone to forget your objective and simply explore in gape-jawed wonder.**

*David Sieks is a contributing editor to* Game Developer. *You can contact him via e-mail at dsieks@arnarb.harvard.edu or through Game Developer magazine.*