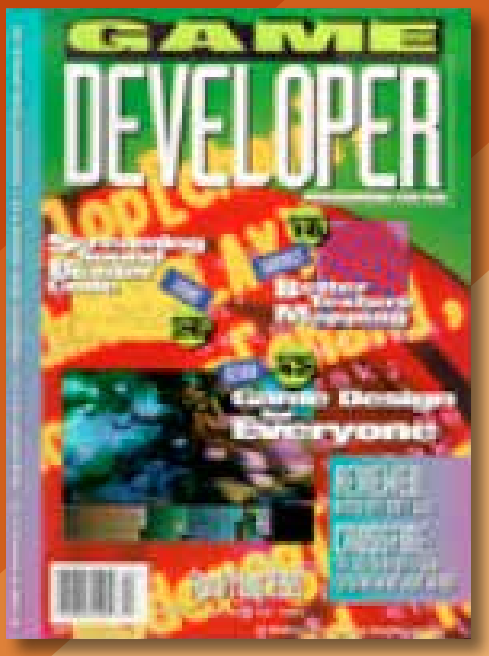
gd

GAME DEVELOPER MAGAZINE

APRIL/MAY 1995



UBM Tech

Delphi is the Answer

Normally I make it a rule not to recommend any programming tool that I haven't used to develop a program from scratch. I'm going to break that rule to recommend Borland's Delphi. I've only used it to develop a program that translated from Celsius to Fahrenheit degrees, but the only reason I'm not altogether committed to Delphi is (I'm blushing as I type this) I've never learned Pascal. Well, I'm going to learn it now. For Windows developers, and for Windows game developers especially, Delphi is so far ahead of the competition it's embarassing.

The world of Windows development has always been one of compromise. Tools such as Visual Basic offer immediate gratification when moving from writing code to seeing your program run, but the performance has always suffered in comparison with compiled code. On the other hand, compiled C++ offers near-optimal performance, but all but the most trivial programs have compilation times measured in minutes, not seconds. Programmer productivity vs. compiled performance. Pleasure for the programmer vs. pleasure for the user. The iron compromise of Windows.

Delphi makes no compromises. Forget the claims of how many hundreds of thousands of lines of code are compiled per minute, the point is that Delphi's speed of compilation is such that the move from editing to debugging is seamless; when your screen flickers you think a background compilation process is being spawned but, in fact, you're seeing the results of the final link.

Delphi is being marketed toward the current marketing darling, corporate developers, and there's no question that it will cut a swath through the ranks of Visual Basic and PowerBuilder users out there. However, game developers have even more to gain by examining Delphi. Why? The huge, relatively untapped Windows game market. With Delphi, you can use WinG to get your images on screen relatively quickly, and you also can get many more compile-link-debug cycles in a programming session. Windows game development inevitably involves hit-and-miss programming challenges (no matter how experienced you are, the Windows API can surprise you), so more cycles means faster development. When you're done, your programs run at compiled speeds.

Also, Delphi abstracts the Windows API. If this abstraction isn't good enough for you, feel free to create your own. Delphi's code reuse infrastructure is excellent, with full object-orientation and the ability to package groups of objects together as functional components. Develop an engine for isometric projection, and you can save it as its own package and use it as a starting point for future games.

Delphi is based on Object Pascal which, as I said, is a new language to me. However, it's very straightforward—C++ it ain't. Although I've already discovered some things that I don't like (the keyword `private` doesn't act the way I think it should, and I can never figure out where semicolons are expected), I've also already come up to speed on the basic structure of the language. You should have no trouble picking it up and, just as importantly, you shouldn't have any trouble finding new programmers fluent in it.

You'll be reading a lot about Delphi in the corporate programming magazines, where it will be primarily lauded for its ability to create database applications. But I'll make a bold prediction—the most successful Windows game of 1996 will be written entirely in Delphi. Until that ships, I'll be busy learning where the stupid semicolons go. ■

Larry O'Brien
Editor

GAME DEVELOPER

Editor Larry O'Brien
gdmag@mfi.com
Senior Editor Nicole Freeman
76702.706@compuserve.com
Managing Editor Nicole Claro
nclaro@mfi.com
Editorial Assistant Deborah Sommers
dsommers@mfi.com
Contributing Editors Alex Dunne
75010.2665@compuserve.com
Chris Hecker
checker@bix.com
David Sieks
dsieks@arnarb.harvard.edu
Wayne Sikes
70733.1562@compuserve.com
Editor-at-Large Alexander Antoniades
sander@mfi.com

Cover Photography Charles Ingram Photography

Publisher Veronica Costanza
Group Director Regina Starr Ridley

Advertising Sales Staff
West/Southwest
Yvonne Labat (415) 905-2353
ylabat@mfi.com
New England/Midwest
Kristin Morgan (212) 626-2498
kmorgan@mfi.com

Marketing Manager Susan McDonald
Advertising Production Coordinator Denise Temple
Director of Production Andrew A. Mickus
Vice President/Circulation Jerry M. Okabe
Group Circulation Director Gina Oh
Circulation Manager Kathy Henry
Circulation Assistant Phil Payton
Newsstand Manager Pam Santoro
Reprints Stella Valdez (415) 655-4269

Miller Freeman
A MEMBER OF THE UNITED NEWSPAPERS GROUP
Chairman of the Board Graham J.S. Wilson
President/CEO Marshall W. Freeman
Executive Vice President/COO Thomas L. Kemp
Senior Vice Presidents H. Vern Packer, Donald A. Pazour, Wini D. Ragus
Vice President/CFO Warren (Andy) Ambrose
Vice President/Administration Charles H. Benz
Vice President/Production Andrew A. Mickus
Vice President/Circulation Jerry Okabe
Vice President/Software Development Division Regina Starr Ridley

More to the Mystery

by Our Readers

Something on your mind? A response to an article, a query about code, a little poem or ditty, perhaps? If you have anything to get off your chest, please do. We'll probably print it here.

Dear Editor:

I would like to offer a few suggestions to supplement Andre LaMothe's article "The Mysterious Mode 13h" (Sept. 1994).

I am a shareware game programmer who has been using Mode 13h for about two years now. I can see at least one obvious ways to speed up the graphics functions Mr. LaMothe described. The first would be to avoid using the palette ports (0x3C7/0x3C8) as much as possible. One blatant way that I can see to do this would be to eliminate reading the port whenever the program needs the value of a specific register. Instead, simply create a structure:

```
typedef struct {
    unsigned char red[256];
    unsigned char grn[256];
    unsigned char blu[256];
    } PaletteStr;
```

Clear the structure in the beginning of the program, and also clear the palette (set all "bucket" values to (0,0,0)) using the write port. Then, whenever the structure is updated, simply update the array value as well. That way, a function can simply look up a value in an array instead of having to write/read to a port. This also streamlines the color cycling function by eliminating the overhead associated with procedure calls (about five processor cycles). Simply rotate the array, then push the whole thing out to the VGA card when the cycle function is done.

Mason McCuskey
via e-mail

MORE READING MATTER

Dear Editor:

I enjoy your magazine, I find it helpful and interesting. I would like to see the price go down or the amount of material in it go up, but I still think that it's worth it. I think it would be enormously helpful if after each article you gave a brief reading list of good books that provide more information about a particular topic. For instance, in the December issue you featured a great article on Mode C (umm... excuse me...Mode X) but I would like to know of a book that would give a beginning-to end breakdown of the technique. This would help those who aren't quite as technically advanced as some, and keep the folks who are advanced happy. Plus the market for computer books is so full of crap that it would be nice to have someone who knows something suggest good books.

Gideon Stocek
via e-mail

Editor Larry O'Brien responds:
Good suggestion! We're running book reviews regularly to try to cut through the crap (see Dean Oisboid's article on page 48). One suggestion: don't buy a book that calls you stupid. If the title's condescending, the text will be as well.

WHERE'S THE CODE?

Dear Editor:

I'm looking for the code for Carl Muller's article "Hitting on Collision Detection" (Sept. 1994). The article states that it is available on CompuServe, but I can't find it. Do you have any more detailed information on exactly where it is, the filename, forum, and so on? I'd also like to know if it's available on any ftp sites.

Leonard Guy
via e-mail

Editor Larry O'Brien responds:
On CompuServe, Go SDFORUM and visit our library. Or set your browser to the /pub/gdmag/src ftp directory of whiz.mfi.com.

Media Vision Picks Up The Pieces

Alex Dunne

"We'll be quiet, but we'll be here," says Robert Brownell, CEO of Media Vision. The company took some recent blows —many of them self-inflicted. Now it's back with a kinder, gentler business sense.

Digital entertainment is often in the media spotlight. Usually we hear about technological breakthroughs, though—rarely are high-tech companies involved in controversies. There have been some notable exceptions, though, and surely Fremont, Calif.-based Media Vision stands out as a Silicon Valley soap opera. Media Vision scraped bottom last summer when it filed for Chapter 11, ironically due to the *lack* of vision that the company—led by former CEO Paul Jain—had. However, this year Media Vision emerged from bankruptcy and is attempting a comeback with the help of new management and capital. The company has refocused on its core technology, audio cards, and is in the process of rebuilding the public's confidence and researching new audio technologies.

The company's problems began back in 1993. Media Vision was well positioned in the red-hot multimedia market, thanks to its Pro AudioSpectrum sound cards and multimedia upgrade kits, yet ongoing price wars with rival sound card makers like Creative Labs and an overextended product line were taking their toll on the company. In October of 1993, Media Vision rolled out a plan to produce CD-ROM games, instantly making it a major player in the game industry.

As a complementary product line, the company's announcement was well received by investors. News of the game venture helped propel the company's stock. Media Vision shares reached $46 in January 1994—barely 14 months after the company went public at $15. Although the stock was generally regarded as overvalued, nobody realized how much so until a few months later.

Greenberg Blows The Whistle

Herb Greenberg, a columnist for the *San Francisco Chronicle*, did some investigating and discovered that Media Vision had falsified sales records and engaged in shady business practices to paint a better picture of the company's finances. According to Greenberg, Media Vision secretly rented warehouses to hide returned merchandise, pressured engineers and sales reps to get defective products out the door, offered huge incentives to distributors and retailers who ordered large quantities of merchandise (known as "channel stuffing"), and most egregiously, doctored shipping papers to charge a large quantity of multimedia upgrade kits to a previous fiscal quarter.

Jain, who some believe was the force behind the company's questionable practices, was a favorite target in Greenberg's columns. One of Greenberg's columns even poked into Jain's personal life, referencing allegations that he illegally used Media Vision's corporate assets to "court women." I bet Jain loathed opening up his morning paper for fear of what he'd read in the business section.

When these stories surfaced, the FBI and the SEC quickly took notice. On May 9, 1994, the two agencies began a probe into securities violations committed by the company. Media

Brownell's humility contrasts his company's brazen past, and he's circumspect about the hard lessons the company has learned with consumers and resellers.

Vision's stock, which had been plummeting in recent weeks due to questions surrounding the company's financial status, closed that day at 2 $^{7}/_{8}$. One week later, Jain and three other corporate officers resigned.

In the wake of Jain's departure, Robert Brownell, the vice president of domestic sales, took over as CEO. Brownell inherited the unenviable position of rebuilding the company, and one of his first actions was selling the company's game publishing arm, which included The Daedalus Encounter and Critical Path, to Virgin Interactive.

Unfortunately for the company, Brownell's move didn't stave off bankruptcy. Two major creditors balked at the company's reorganization plan, so on July 25th, 1994, Media Vision filed for Chapter 11. The clincher followed: the company admitted that its $20 million profit for 1993 wasn't quite accurate. In fact, there was no profit for the previous year: the company actually lost $99 million. Media Vision attributed the revision in numbers to unaccounted product returns and marketing costs, improperly recorded sales, and doubtful accounts.

Since it came clean with investors last summer, Media Vision has taken steps to rebuild itself. In August, it secured a $10 million credit line from Trust Company of the West (TCW) to help fill $24 million in backlogged orders and prepare for heavy demand during the Christmas shopping season.

On December 30th, Media Vision emerged from Chapter 11. Its stock, which just a year earlier had been trading in the 40s, was canceled and worthless. To cover previous debts, the company issued 20 million shares of new common stock, of which TCW will own 43%. Two TCW officers sit on Media Vision's board of directors. The company, once composed of close to 500 employees, is about half that size now. Media Vision has also entered into an agreement with the SEC, according to which the agency will recommend that no enforcement action be taken against the company, as long as Media Vision cooperates with the SEC and the Dept. of Justice.

From the CEO: Brownell's Plans To Rebuild

I recently spoke with Robert Brownell, and he explained Media Vision's plans. His humility contrasts his company's brazen past, and he's circumspect about the hard lessons the company has learned with consumers and resellers.

"Our position is more one of controlled growth. We're not just chasing a growth curve to see how much in sales we can ring up. We're very cautious when dealing with the reseller channel that we don't put too much product in, [rather] that we put the right product in. We don't try to give [resellers] eight kits when we know only two of them are going to sell. We're not in that mode anymore. I guess some of the competitors still do that," Brownell commented.

Brownell on the practice of channel stuffing: "That happens everywhere. [If] you continue to throw out new products and... you don't deal with [your older products] as rapidly as you should, then you have a problem. We monitor the sell-through at the retail level very closely, to make sure that we don't put our products in that position. And we don't do deals with the reseller channel that could put us in that position. We just say, 'Buy what you think you can sell. Here's our pricing. If this works for you, then great. We'll support you.'"

What are Brownell's goals for Media Vision in 1995? "We haven't publicly stated our dollar goal, and we probably won't until the end of the first quarter. But [with respect to] my goals for the company, the first... was to get through bankruptcy. To get our house in order. And now we're in that position. We have developed a plan to go forward, [and] we need to execute on that plan... 1995 should be a quiet year for us."

Brownell has stated that Media Vision will get back to its core market. I asked him to elaborate on this plan: "Our core technology is audio, graphics, and actually, video. One of the areas that we've been working on over the last several years is waveguides. We have one of the licenses out at Stanford and we think we're the farthest along, and we have a working chip at this time. We hope that this will translate into some products in the fourth quarter, but I don't think it will have a major impact until next year. We feel that's our best shot, and we want to build our company around it. It will differentiate us in the market."

It was encouraging to hear that waveguides is an integral part of the

company's plans. Waveguides is a software technology that was developed at Stanford University's Center for Computer Research in Music and Acoustics. It uses algorithms to mimic the subtle characteristics of musical instruments and human voices.

"[Waveguides] is professional audio at consumer prices," Brownell explained. "You can develop sounds and you can create new sounds that have never been developed before. If you wanted to know what a 10-foot-long trombone sounded like, you could do it. A guitar with a 15-foot-long neck. You can actually recreate the [sound of] vocal chords and how they interact with the mouth cavity. When you hear a clarinet, you hear the [player's] breath. We think this is the next standard of sound. We'll be showing it to game developers soon."

Brownell also indicated that due to Media Vision's past financial troubles, the company was no longer pushing for a VESA standard for audio cards (see "Sounding Out the VESA Audio Standard" by Jon Burgstrom, June 1994).

Does Media Vision have any plans to reenter the game development market?

"No. I shouldn't say never, but it's something that takes a lot of focus and attention. And because of the fact that it's not our core strength, and that we don't devote 100% of our time to it, it's a difficult business. It's a 'hits' business. The cost of developing [games] is so high—it requires a lot of cash and investment before you get a payoff. I feel that it's better left to people who are devoting 100% of their attention to it. It's a million dollars to develop a title—that's before promotion costs.

"The start we got off to in that business wasn't very good. We didn't have enough to sustain it. [Of] our existing titles, two or three were good, and the rest were O.K. I liked a lot of the [games] that were coming down the pipe, but it would have been another 12 to 24 months before they started to show any returns. And, at the same time, you have to continue to develop because you can't stand still. So I really do believe it's best left to people who just deal with software."

Brownell had some interesting insights into the multimedia upgrade market, in which he sees a second round of upgrade kit purchasing. This coming round of purchasing will be fueled by people who already bought an upgrade kit or PC containing a single or double-speed CD-ROM drive and standard 8-bit or 16-bit sound card, and who would like to move up to a quad-speed unit and a superior sound card. Brownell doesn't see this second-round market growing as explosively as the first round of kit sales.

What's his take on Media Vision's competitor, Creative Labs?

"I know that they have a lot of product out in the channel. Again, we're not chasing [Creative Labs' projected] growth curve. They said they were going after a billion dollar [in sales] year, and that's too risky for us right now. If we build good products, and make it easy for the customer to use them, ...do the right price points, the sales will come. And whatever [the sales] are, they are. If you do a good job and you support the customers, your sales will increase. But if you try to force it, and think that no matter what you put in the kit is going to sell at whatever price point, you're only fooling yourself in the long run. It will be interesting to see how the industry shakes out. We'll be quiet, but we'll be here. Especially through '95. We gotta prove to people that we can do it. Deliver what we said we were going to deliver on."

What has Brownell heard about ex-CEO Paul Jain?

"Nothing. I haven't really talked with him since he left. He might have already started another company, but I'm not sure."

There's hope for this company. ■

Alex Dunne is contributing editor for Game Developer *magazine. Contact him via e-mail at 75010.2665@compuserve.com or through* Game Developer.

Hair in a Can it Ain't

Nicole Claro

Animation, animation, animation! It seems like that's what it's all about. But it's also about upgrades, design, textures wacky and wonderful and the hairdo to end all hair-dos (Ted Danson, eat your heart out).

One of the greatest challenges facing game today's game animators is not how to create amazing graphics, or how to render realistic morphs, or even how to incorporate believable live-action video into their games. There's a pursuit far thornier than any of these, and that is...hair.

Yes, hair. Whether the character needs a Mike-Brady perm, a floppy toupee, or a purple mohawk, designers have always had difficulty creating luscious head-topping or body-covering tresses that are true-to-life. Until now, that is. Alias CompuHair will put an end to bad hair days for three-dimensional characters forever. The new product from Alias uses an advanced computer technique called particle systems, which uses the power of Silicon Graphics workstations to render real-looking hair and fur.

Users can customize the hair using options for color, transparency, length, curliness, and thickness. The Alias Digital OptiF/X system, the main component of Alias PowerAnimator 6.0, integrates the Alias particle system generator to create interactions between particles and NURBS or polygonal-based models in the scene. It's this integration that lets you combine hair and fur with other three-dimensional modeling and rendering elements like wind, gravity, and light, to create waving hair, shadowed hair, glowing hair, and the like.

Wait, there's more! If you want to find out even more about this, simply stay up until 1:00 a.m. sullenly driving the channels with your remote until you see the infotisement. Actually, you can join the Alias Hair Club for Animators by accessing Alias's World Wide Web server at www.alias.com. Perhaps you'll meet the organization's president who, I understand, is also a client.

For More Information Contact:
Alias Research Inc.
110 Richmond St. E.
Toronto, Ont., Canada M5C-1P1
Tel: (416) 362-9181

Waves of Stucco and Velvet

Sure, hair's important—who hasn't been hair-obsessed at one time or another (especially during adolescence)? But there are lots of things that have nothing to do with live characters that can often make or break a game. Fractal Design's Really Cool Textures recently came on the market as add-on packages for use with Fractal Design Painter and Fractal Design Sketcher.

The company's newest series increases the number of paper grains and patterns available to users of Painter and Sketcher. You can create a texture and use it as a background or link it to any natural-media tool to allow you to paint with textured ink or paint. The series comprises Miles of Tiles, Walls and Reliefs, Grains and Weaves, and Patterns and Nature. Really Cool Textures costs $29.99.

For More Information Contact:
Fractal Design Corp.
335 Spreckels Dr.

Aptos, Calif. 95003
Tel: (408) 688-5300

Quickdraw and More

Electric Image Inc.'s ElectricImage Animation System now supports Apple's quickdraw three-dimensional API and metafile format. With this technology, users can view their animation projects in real time, as they are creating them. Accelerated three-dimensional graphics can only increase the speed of the user interface, the company says.

Once available only for high-end workstations, the new API and three-dimensional cards will soon be open to all customers. Electric Image will release two versions—one for Power Macintosh and one for Macintosh. Both versions import, render and animate objects from multiplatform modeling programs and both include sync sound animation, deformations, and various plug-ins. ElectricImage Power Macintosh 2.1 and Macintosh 2.0 sell for $7,495. Owners of ElectricImage 2.0 (Macintosh version) can upgrade to the Power Macintosh 2.1 versions for $495.

For More Information Contact:
Electric Image Inc.
117 E. Colorado Blvd., Ste. 300
Pasadena, Calif. 91105
Tel: (818) 577-1627

Toonz Galore

Microsoft has just released version 3.5 of SoftImage Toonz (the software formerly known as Creative Toonz—not to be confused with the rock star) for the Silicon Graphics platform. The product includes a complete range of functions based on traditional cel-drawing processes, such as audio input to scanning, pencil test, set palette, special effects, and compositing. By adding separate modules, you can customize SoftImage Toonz for scanning, ink and paint, and rendering.

Version 3.5 includes enhanced versions of many of the features available on past versions as well as new modules and utilities, and several major bug fixes. SoftImage Toonz 3.5 features an icon-based, simpler user interface; X-sheet, an exposure sheet based on the ones used in traditional animation; a camera stand, which adjusts cels and layers to size and background, that exists in the same interface as the X-sheet; and resolution independence, which lets users select film, video, or HDTV output. SoftImage Toonz 3.5 costs approximately $16,995, with various modules available at extra cost.

For More Information Contact:
Microsoft Inc.
1 Microsoft Wy.
Redmond, Wash. 98052-6399
Tel: (206) 882-8080

Ooh, Cyberspace!

Autodesk has upgraded its Cyberspace Developer Kit (CDK). Release 2 is a Windows NT/32s-based upgrade of the toolset for three-dimensional visualization and simulation. CDK Release 2 features a comprehensive code set that incorporates 170 C++ classes and more than 1,400 functions that lets professional users, such as game developers, create interactive, three-dimensional environments. It's rendering-solution independent and can be extended to support future hardware- and software-based rendering methods. CDK Release 2 costs $1,995.

For More Information Contact:
Autodesk, Inc.
2320 Marinship Way
Sausalito, Calif. 94965
Tel: (800) 879-4233

Further Rendering

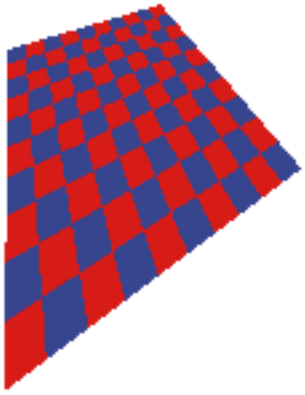
And in other upgrade news, RenderMorphics has released version 2 of Reality Lab, its three-dimensional API. With full cross-platform integration and compatibility, Reality Lab 2 provides new special effects options and a feature set that includes depth cueing, projected shadows, and more. The functionality of version 1 is further enhanced through Immediate mode, a module that allows low-level control over polygon lighting and rendering through direct access to vertices and normals. This improves custom lighting and transform effects, warping, twisting, and bending of object, and procedurally defined objects. Reality Lab 2 supports Windows, DOS, System 7 and UNIX and contains specific Pentium optimization, which ensures that it is even faster than version 1.

For More Information Contact:
RenderMorphics Ltd.
Unit 15, The Turnmill
63 Clerkenwell Rd.
London EC1M 5NP U.K.
Tel: 44 (0) 71 251 4411

Nicole Claro is managing editor for Game Developer *magazine.*

Perspective Texture Mapping Part I: Foundations

Figure 1. Textured Checkerboard



If there is one technical feature today's high-performance three-dimensional games must have, it is texture mapping. The technique of texture mapping stretches across almost every genre of game, from role-playing games like Ultima Underworld and System Shock, through simulators like Indy Car Racing and Wing Commander III, to action games like Doom and Descent.

Given its popularity, you'd think there would be a wealth of information available on how to actually write your own perspective texture mapper. You'd be wrong.

When I was researching this article (actually, when I was trying to figure out if an article on perspective texture mapping was even needed), I looked high and low for intuitive descriptions and working sample code, but not much exists. Most articles on the Internet describe affine texture mapping, and the few perspective texture mapping articles I did find on x2ftp.oulu.fi, an excellent game programming ftp site managed by Jouni Miettunen, use overly complicated descriptions and aren't accompanied by working code. Even old standbys, like *Computer Graphics: Principles and Practice* (Addison-Wesley, 1992) by James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes (commonly called Foley and van Dam, much to the chagrin of Feiner and Hughes, I'm sure) and the bible of texture mapping, *Digital Image Warping* (1990, IEEE Computer Society) by George Wolberg, are woefully inadequate if you actually want to write a texture mapper, especially one fast enough to be compelling for games.

I'm going to address this lack of documentation in this and the second part of this article. First, using nothing more than basic algebra and geometry, I'll show you an easy-to-understand mathematical foundation, for how and why perspective texture mapping works. I'll also provide sample code to implement the naive algorithm. In the second installment, we'll speed it up to interactive performance.

Assumptions, Definitions, and Concepts

If we want to cover everything in two articles, we're going to have to move pretty fast. To do this, I need to assume you know a bit about three-dimensional graphics. If you don't know how object space, world space, view space, and screen space interact, or you don't know what those terms mean, you should probably pick up a book like Foley and van Dam before reading this article.

The term "texture mapping" describes a whole family of techniques, but for these articles, we'll define texture mapping as drawing a planar polygon as if a bitmap was glued to the polygon's face. This bitmap goes through the same transforms (or at least looks like it does) as the polygon, so if we view the polygon almost edge on, the bitmap, or texture, will look like it's edge on as well. Figure 1 shows a checkerboard viewed at an angle. You can see how the squares get smaller as they recede, just as you'd expect.

To accomplish this mapping, we associate a texture bitmap with each polygon and texture coordinates with each vertex of the polygon. In addition

to the normal (x,y,z) triplet to define a vertex in three dimensions, we specify the texture coordinates u and v. These coordinates are two-dimensional coordinates into the texture bitmap, and the pixels in this bitmap are sometimes called texels.

To make things easy to visualize, our diagrams and equations will be in two dimensions—think of working in a slice through the three-dimensional space—but our results extend easily into three dimensions.

Perspective Projections

Most three-dimensional game graphics are based on perspective projections. Perspective projections make distant objects seem smaller than closer objects and distort angles so scenes look realistic.

The basic equation for the perspective projection uses similar triangles that share a vertex at the origin (the viewpoint). If we take the point (x_0,z_0) (ignore the u coordinates for the time being) and project it onto the dashed vertical z=d line in Figure 2 to give us (x_0',d), the equation for the relationship between these two points is:

$$\frac{x'_0}{d} = \frac{x_0}{z_0}$$

In other words, the ratio of the height of the triangle formed by ((0,0), (x_0',d), (0,d)) to the length of its base is the same as the ratio of the height of the triangle formed by ((0,0), (x_0,z_0), $(0,z_0)$) to the length of its base. If we assume d=1 for the current example, and generalize this equation to all unprojected points (x,z), we get:

$$x' = \frac{x}{z} \qquad (1)$$

If we view the z=d line as the one-dimensional equivalent of the two-dimensional screen plane (pretend you're looking down on the plane from above, so you can only see it as a line), Equation 1 says we can generate screen coordinates (x´ for values of x) by dividing the unprojected object coordinates by their z values. This is the perspective projection in its essence.

Mapping Direction

In three-dimensional graphics, we consider transforming from object to screen coordinates moving "forward," so Equation 1 is called a forward mapping—it projects the source polygon forward onto destination pixels. To use a forward mapping for texturing a polygon, you step along the polygon in object space and project each generated point forward to a destination pixel position. Forward mappings don't work very well for texture mapping, however, because it's hard to be sure how far to step in the source so that the projected coordinates don't skip or overwrite any pixels in the destination.

Backward mappings, on the other hand, allow us to step in screen space, processing each pixel exactly once. If we manipulate Equation 1 to give us a backward mapping from x´ to x we get:

$$x = x'z \qquad (2)$$

This tells us we can generate values of x from values of x´ if we multiply x´ by z. We can easily generate the desired u texture coordinate once we have x, but first we must find the correct z to feed into

Chris Hecker

Little has been written on perspective texture mappers, an invaluable feature of any high-performance game. This month, Chris Hecker fills the void with the first of a two-part article on the subject.

Equation 2 (we already know x´ because it's the current pixel we want to write).

It would be great if we could generate z values directly from x´ values using a simple linear interpolation. We often use linear interpolations in graphics in the form of digital differential analyzers (DDAs), and fixed and floating-point interpolations, but we know linear interpolation is only accurate when we are interpolating a linear equation. Let's explore the relation between x´ and z to see whether they are linear with respect to one another, which in turn will tell us if we can use linear interpolation to generate z from x´.

A linear equation is any equation of the form:

$$y = AX + B \quad (3)$$

for any real values of A and B (this is called the slope-intercept form, where A is the slope of the x,y line, and B is the y-intercept, or value of y when the line crosses the y axis). That is, as x changes by a constant amount, y changes by a constant amount proportional to the change in x.

To find the relationship between x´ and z, we first take the equation for the unprojected line in object space, x = Az + B. The actual values of the constants A and B are based on the endpoints of the line segment, and are irrelevant to this derivation. Next, we substitute this into Equation 2 to get an equation in z and x´, and solve for z:

$$Az + B = x'z$$
$$B = z(x'-A)$$

and finally:

$$z = \frac{B}{x' - A} \quad (4)$$

Equation 4 is definitely not a linear equation with respect to x´, so we can't directly compute z incrementally from values of x´. However all is not lost, because a little algebraic manipulation gives us:

$$\frac{1}{z} = \frac{1}{B}x' - \frac{A}{B} \quad (5)$$

Equation 5 is a linear equation with respect to x´. The only problem is, it's a linear equation of 1/z with respect to x´, not z itself! We can use Equation 5 to linearly interpolate values of 1/z and take the reciprocal at each pixel to get the real value of z. In other words, we can linearly interpolate 1/z and divide x´ by 1/z to generate values of x according to Equation 2. These values of x allow us to compute values of u that we can use to look up the correct color from the texture bitmap to store in x´. Voila, perspective texture mapping.

It turns out we can compute u directly instead of computing x, saving a step and simplifying our lives. By definition in Equation 1, x/z is linear in screen space (it's actually equal to screen space, which is about as linear as you can get!). Just as x and z are linear with respect to each other because the object is planar (or linear, in Figure 2), u and x are linear with respect to each other for the same reason. Well, if x/z is linear in screen space, and x is linear with u, then u/z is linear in screen space as well (you can prove this to yourself by playing around with Equations 1, 3, and 5). Instead of dividing x/z by 1/z to generate x coordinates that we then use to solve for u coordinates, we can interpolate u/z and divide it by 1/z to generate the u values directly.

Affine texture mapping ignores these results and linearly interpolates u and v in screen space without the divide. This results in funky warping, but for some polygons it's not too bad (and because there's no divide it has the potential to be a lot faster). A comparison is beyond the scope of this article, but you can find affine texture mappers on x2ftp.oulu.fi, which I mentioned previously.

Our Story So Far

Let's take a break, sum up our results to this point, and outline a simple algorithm to perspective texture map the line segment in Figure 2.

We've shown that 1/z and u/z are linear in screen space, so the algorithm for texture mapping Figure 2 goes like this:

- Project the object vertices into screen space, giving x´ = x/z and u´ = u/z.
- Let z´ = 1/z at each vertex.
- Linearly interpolate u´ and z´ between x_0´ and x_1´, stepping x´ by 1 pixel each loop.
- At each pixel x´, calculate u by u´/z´, and use u to fetch the correct texel.
- Write the texel to the destination at x´.

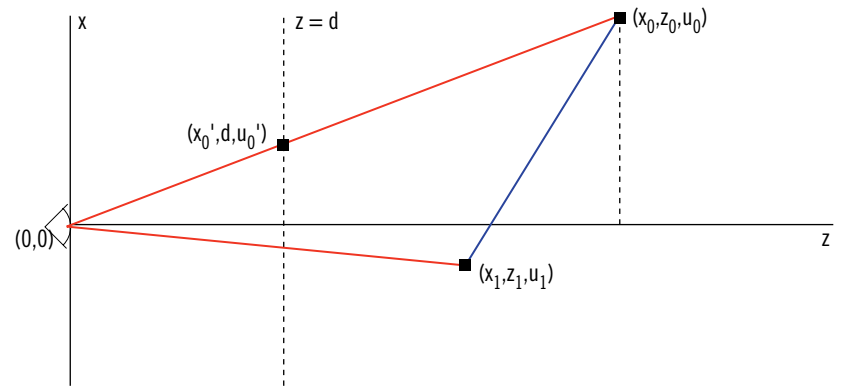
The proofs for y and v are analogous to those for x and u, so this is all there is to writing a three-dimensional perspective texture mapper.

Interpolation Breakdown

In the simplified algorithm I've outlined, we linearly interpolated u´ and z´ over the length of the scanline. Each linear interpolation usually involves these steps:

- Figure out the start and end values of the interpolants (in Figure 2, u_0´,z_0´ and u_1´,z_1´, respectively).

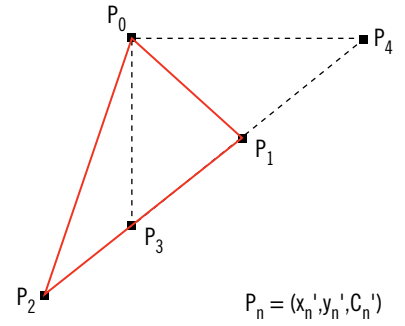
Figure 2. Perspective Projection



- Calculate the amount each changes as it moves from start to end ($u_1' - u_0'$ and $z_1' - z_0'$).
- Divide the change by the distance over which you want to interpolate ($x_1' - x_0'$) to get each step.
- Increment from the start to the end by this step.

This is a fair amount of work, and if we plan to rasterize polygons like the triangle shown in Figure 3, we have even more work to do. We need to interpolate at least 1/z, u/z, and v/z (and possibly one or three colors), and if we first calculate the interpolants down each edge, and then calculate new ones when we get to each scanline we will soon get lost in a sea of interpolants going in all sorts of directions. Luckily, there is a better way.

Figure 3. Calculating Gradients



It just so happens that the increments in each linear interpolant for a single step in x or in y are constant across the whole polygon. This is a very important and very cool result because it means we can calculate these increments—called the gradients—once and never need to worry about calculating interpolants again during rasterization. In other words, when we want to rasterize a polygon, we calculate the gradients in x and in y for each parameter at the very beginning, and every time we step in x or y or both we just add in the appropriate gradients. When we get to a scanline we want to draw, we don't need to calculate linear interpolations for all of our parameters as we step across the scanline in x, because we already have their gradients with respect to x sitting around! In the same vein, stepping down an edge is simply some combination of the gradient in x and the gradient in y. (If you think about it, this also means you only need to interpolate the parameters down one edge. Ponder that one for a while.)

To show how gradients are calculated, let's use the triangle $P_0P_1P_2$ in Figure 3. Each vertex has a screen space x and y associated with it (x', y'), but in addition there is an arbitrary parameter, c', which could be color for Gouraud shading or 1/z, u/z, or v/z for perspective texture mapping. It is any parameter we can linearly interpolate over the surface of the two-dimensional (screen space) triangle.

Given this triangle, let's figure out how the parameter c' changes if we hold y constant and step in x. We will use the point P_4 in our construction (P_3 and P_4 are both on the P_1P_2 line in Figure 3). It is clear that $y_4' = y_0'$, and we can derive the other coordinates for P_4 using the line equations:

$$\frac{x_1' - x_2'}{y_1' - y_2'} = \frac{x_4' - x_2'}{y_4' - y_2'}$$

and:

$$\frac{c_1' - c_2'}{y_1' - y_2'} = \frac{c_4' - c_2'}{y_4' - y_2'}$$

Substituting y_0' for y_4' and solving for the various coordinates gives us:

$$y_4' = y_0'$$

$$x_4' = \left(\frac{x_1' - x_2'}{y_1' - y_2'}\right)(y_0' - y_2') + x_2'$$

and:

$$c_4' = \left(\frac{c_1' - c_2'}{y_1' - y_2'}\right)(y_0' - y_2') + c_2'$$

Next, refer to Figure 5 to compute the difference in c' (called dc') as it moves from P_0 to P_4 with Equation 6.

The analog for c' with respect to y as we move from P_0 to P_3 is also shown in Figure 5, in Equation 7. (Notice the denominators: dx = -dy.)

The values dc'/dx and dc'/dy are called the gradients for the parameter; dc'/dx is the gradient with respect to x and dc'/dy is the gradient with respect to y. We can calculate the gradients for 1/z, u/z, and v/z with respect to both x and y at the top of our texture mapper and never need to calculate them again during the rasterization of this polygon.

Our new texture mapping algorithm looks like this:

- Project the object vertices into screen space, giving x', y', $u' = u/z$, $v' = v/z$, and $z' = 1/z$.
- Calculate the gradients in x and y for u', v', and z'.
- Linearly interpolate down each edge and across each scanline using the gradients.
- At each pixel, calculate u by u'/z' and v by v'/z', and use u and v to fetch the correct texel.
- Write the texel to the destination at x', y'.

The only thing we're missing is a consistent fill convention to make sure we light the correct destination pixels as we rasterize the polygon. Once we have a fill convention, we can guarantee polygons will abut properly and we won't have any skipped pixels (dropouts) or overwrites at the edges.

Conventional Wisdom

A fill convention is a set of rules that describes how to light pixels in the screen under various edge conditions. The first step towards implementing a fill convention is defining exactly which pixels we want lit when a polygon is rasterized. Figure 4 shows the raster grid of the display, with pixel centers marked with black dots.

We will define what's called a top-left fill convention. Top-left refers to the tie breaking rule used when the edge of a polygon lands exactly on a pixel center; if the edge is a top or a left edge, the pixel is in the polygon, if it's a right or a bottom edge, the pixel is considered out. You can see this convention in Figure 4. If the red edge is shared by the blue and the yellow polygon, they will not light any of the same pixels. The horizontal red edge is the top of the yellow polygon, so the pixels are considered members of that polygon. In contrast, the horizontal red edge is a bottom edge of the blue polygon, so the pixels are not lit. All

other pixels—those not intersected on pixel centers—are lit if they are "strictly in" the polygon. In other words, the pixel center must be completely inside the edge for the pixel to be lit. In contrast with Figure 4, real edges are infinitely thin, so the pixel center is either out, intersected exactly, or in.

The next step is to define the fill convention mathematically. A top-left fill convention is defined by the ceiling function for the left and top edges, and the ceiling-1 of the right and bottom edges. (The ceiling function bumps a fractional number up to the next integer unless it's already an integer, in which case the number stays the same.) We'll be stepping in y to generate scanlines, so the equation for generating x coordinates from y coordinates as we step from P_0 to P_2 in Figure 3 is:

$$x = \left(\frac{x_2' - x_0'}{y_2' - y_0'}\right)(y - y_0') + x_0'$$

We apply the ceiling function to this equation to give us integer raster values for a given y:

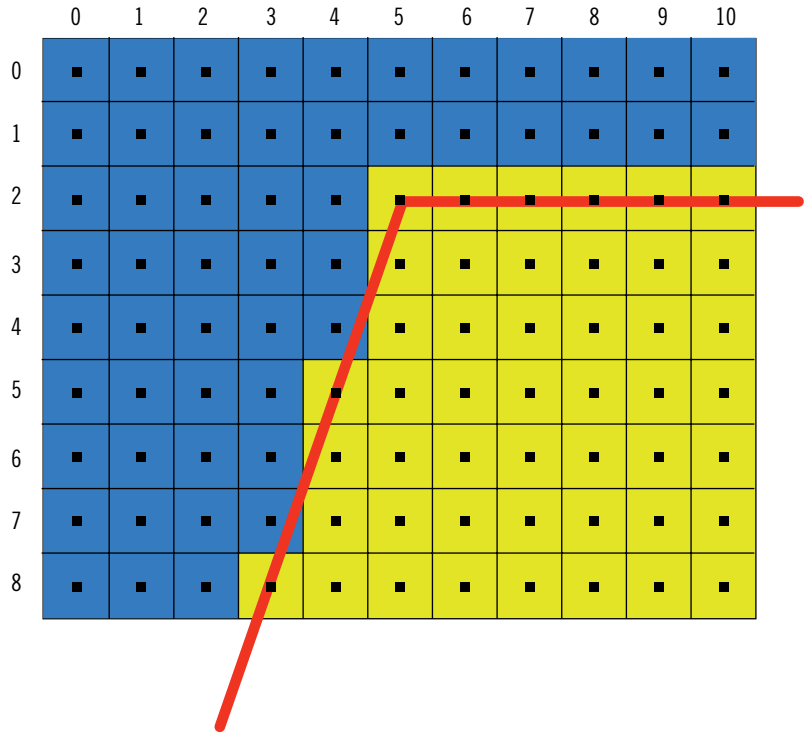
$$x_{\text{int}} = \left\lceil \left(\frac{x_2' - x_0'}{y_2' - y_0'}\right)(y - y_0') + x_0' \right\rceil$$

If our starting coordinates are real numbers instead of integers, we need to apply our convention to the y coordinate as well to generate the initial y value:

$$y_{0\,\text{int}}' = \lceil y_0' \rceil$$

On a number of scanlines in Figure 4, the real edge—the line on which we're interpolating our parameters—differs from the starting pixel center by some small amount. Pixels in the display are not points, they're actually boxes with an area around the pixel center (the pixel center is the integer coordinate, and the box extends 0.5 pixels to each side), and when stepping from pixel to pixel we want to make sure we step from one pixel center to the next. If we don't step on pixel centers, our textures will appear to swim as our polygon rotates because we aren't sampling from the same place in the pixel each time. Also, when reading from what are essentially random places in each pixel it's possible to generate texture coordinates outside the texture bitmap (which could crash our program).

Figure 4. Fill Conventions



Find Your Center

Given that we want to sample the texture from the exact pixel center, we need to make sure our interpolants are prestepped on each scanline by the difference between the real edge and x_{int}. If we do this correctly, our texture mapper will never read outside the texture (assuming the texture coordinates are valid in the first place, of course), and our textures will not swim as our polygon moves around the screen. Also, we won't get the "hairy texture" artifacts you see in a rasterizer that doesn't step on pixel centers, where lines in the texture that should be straight come out with little notches and pimples.

Figure 6 shows a close-up of a group of pixels. To start rasterizing, we must first step our edge to the point A. This involves an x and y prestep for our interpolants, marked with dotted lines. Now, we can precalculate each parameter's step in y and in x for a single scanline step in the screen using the gradients we calculated beforehand, so each time we move from one scan to the next, we just add each step to its interpolant to find the new value. When it's time to draw a scanline, we must step to the first pixel center. Figure 6 shows this step as a dotted line at each scanline.

Figure 5. Equations 6 and 7

$$\frac{dc'}{dx} = \frac{c_4' - c_0'}{x_4' - x_0'} = \frac{(c_1' - c_2')(y_0' - y_2') - (c_0' - c_2')(y_1' - y_2')}{(x_1' - x_2')(y_0' - y_2') - (x_0' - x_2')(y_1' - y_2')} \quad (6)$$

$$\frac{dc'}{dy} = \frac{c_3' - c_0'}{y_3' - y_0'} = \frac{(c_1' - c_2')(x_0' - x_2') - (c_0' - c_2')(x_1' - x_2')}{(x_0' - x_2')(y_1' - y_2') - (x_1' - x_2')(y_0' - y_2')} \quad (7)$$

All this prestepping probably sounds expensive, but there is a way to do it that requires no extra multiplies per scanline. We'll discuss this in more detail next month.

Summary and Random Notes

This article is too long already, but there's still plenty we haven't discussed.

First, we didn't talk about all the special cases where linearly interpolating the texture coordinates actually is correct, like walls and floors. A close examination of the math above will show you why this is true (hint: look at the gradients for the 1/z term). Games like Doom use this to speed up their texture mappers at the expense of not allowing arbitrarily oriented polygons. There's lots of information covering these techniques on the Internet.

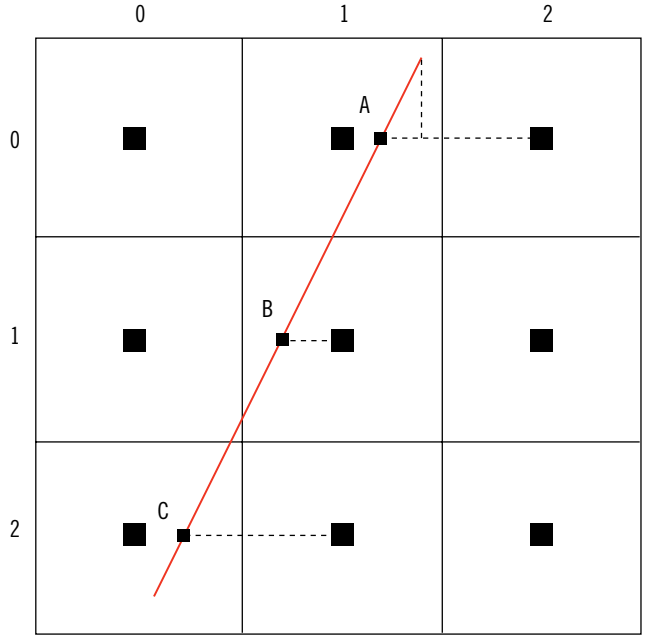
We also didn't discuss antialiasing or homogeneous coordinate systems. *Digital Image Warping* is a great resource for antialiasing and image resampling, while Foley and van Dam cover homogeneous coordinates.

Even considering what we missed, we certainly covered a lot of material in a small space, and I encourage you to reread this article with a piece of paper in hand and try to prove the various results for yourself.

The sample code included with this article implements the perspective texture mapping algorithm. It is a high quality implementation with one small problem: it's a bit slow, doing a divide and two multiplies per pixel. In the next column I'll show how to optimize this code, which will give you a production quality perspective texture mapper you can just plop right in your game engine. ■

Chris Hecker wishes he had a Ph.D. in mathematics so he didn't have to struggle with the derivation of the equation for the area of a triangle whenever he wanted to use it. In the meantime, he can be reached at checker@bix.com or through Game Developer *magazine.*

Figure 6. Pixel Centers



Listing 1. Perspective Texture Mapper

```
#include<windows.h>
#include<math.h>

struct POINT3D {
    float X, Y, Z;
    float U, V;
};

void TextureMapTriangle( BITMAPINFO const *pDestInfo,
        BYTE *pDestBits, POINT3D const *pVertices,
        BITMAPINFO const *pTextureInfo,
        BYTE *pTextureBits );

/******** structures, inlines, and function declarations **********/

struct gradients {
    gradients( POINT3D const *pVertices );
    float aOneOverZ[3];              // 1/z for each vertex
    float aUOverZ[3];                // u/z for each vertex
    float aVOverZ[3];                // v/z for each vertex
    float dOneOverZdX, dOneOverZdY;  // d(1/z)/dX, d(1/z)/dY
    float dUOverZdX, dUOverZdY;      // d(u/z)/dX, d(u/z)/dY
    float dVOverZdX, dVOverZdY;      // d(v/z)/dX, d(v/z)/dY
};

struct edge {
    edge(gradients const &Gradients,
        POINT3D const *pVertices,
        int Top, int Bottom );
    inline int Step( void );

    float X, XStep;                  // fractional x and dX/dY
    int Y, Height;                   // current y and vert count
    float OneOverZ, OneOverZStep;    // 1/z and step
    float UOverZ, UOverZStep;        // u/z and step
    float VOverZ, VOverZStep;        // v/z and step
};

inline int edge::Step( void ) {
    X += XStep; Y++; Height--;
    UOverZ += UOverZStep; VOverZ += VOverZStep;
```

Listing 1. Perspective Texture Mapper (Continued on p. 25)

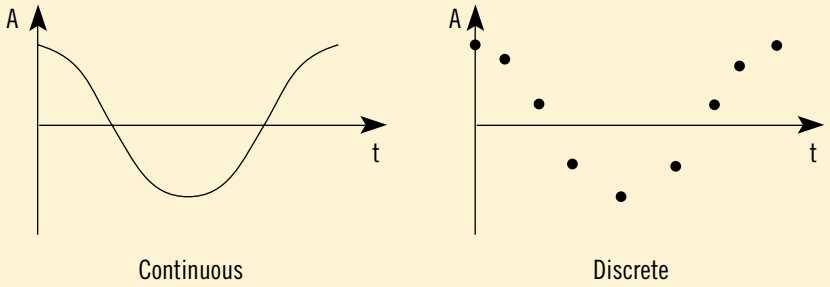
```
            pLeft,pRight,pTextureInfo,pTextureBits);
        TopToMiddle.Step(); TopToBottom.Step();
    }

    Height = MiddleToBottom.Height;

    if(MiddleIsLeft) {
        pLeft = &MiddleToBottom; pRight = &TopToBottom;
    } else {
        pLeft = &TopToBottom; pRight = &MiddleToBottom;
    }

    while(Height--) {
        DrawScanLine(pDestInfo,pDestBits,Gradients,
            pLeft,pRight,pTextureInfo,pTextureBits);
        MiddleToBottom.Step(); TopToBottom.Step();
    }
}

/********** gradients constructor **********/

gradients::gradients( POINT3D const *pVertices )
{
    int Counter;

    float OneOverdX = 1 /(((pVertices[1].X - pVertices[2].X) *
            (pVertices[0].Y - pVertices[2].Y)) -
            ((pVertices[0].X - pVertices[2].X) *
            (pVertices[1].Y - pVertices[2].Y)));

    float OneOverdY = -OneOverdX;

    for(Counter = 0;Counter < 3;Counter++) {
        float const OneOverZ = 1/pVertices[Counter].Z;
        aOneOverZ[Counter] = OneOverZ;
        aUOverZ[Counter] = pVertices[Counter].U * OneOverZ;
        aVOverZ[Counter] = pVertices[Counter].V * OneOverZ;
    }

    dOneOverZdX = OneOverdX * (((aOneOverZ[1] - aOneOverZ[2]) *
            (pVertices[0].Y - pVertices[2].Y)) -
            ((aOneOverZ[0] - aOneOverZ[2]) *
            (pVertices[1].Y - pVertices[2].Y)));
    dOneOverZdY = OneOverdY * (((aOneOverZ[1] - aOneOverZ[2]) *
            (pVertices[0].X - pVertices[2].X)) -
            ((aOneOverZ[0] - aOneOverZ[2]) *
            (pVertices[1].X - pVertices[2].X)));

    dUOverZdX = OneOverdX * (((aUOverZ[1] - aUOverZ[2]) *
            (pVertices[0].Y - pVertices[2].Y)) -
            ((aUOverZ[0] - aUOverZ[2]) *
            (pVertices[1].Y - pVertices[2].Y)));
    dUOverZdY = OneOverdY * (((aUOverZ[1] - aUOverZ[2]) *
            (pVertices[0].X - pVertices[2].X)) -
            ((aUOverZ[0] - aUOverZ[2]) *
            (pVertices[1].X - pVertices[2].X)));

    dVOverZdX = OneOverdX * (((aVOverZ[1] - aVOverZ[2]) *
            (pVertices[0].Y - pVertices[2].Y)) -
            ((aVOverZ[0] - aVOverZ[2]) *
            (pVertices[1].Y - pVertices[2].Y)));
    dVOverZdY = OneOverdY * (((aVOverZ[1] - aVOverZ[2]) *
            (pVertices[0].X - pVertices[2].X)) -
            ((aVOverZ[0] - aVOverZ[2]) *
            (pVertices[1].X - pVertices[2].X)));
}

/********** edge constructor **********/

edge::edge( gradients const &Gradients,
        POINT3D const *pVertices, int Top, int Bottom )
{
    Y = ceil(pVertices[Top].Y);
    int YEnd = ceil(pVertices[Bottom].Y);
    OneOverZ += OneOverZStep;
    return Height;
}

void DrawScanLine( BITMAPINFO const *pDestInfo,
    BYTE *pDestBits, gradients const &Gradients,
    edge *pLeft, edge *pRight,
    BITMAPINFO const *pTextureInfo, BYTE *pTextureBits );

/******** TextureMapTriangle **********/

void TextureMapTriangle( BITMAPINFO const *pDestInfo,
        BYTE *pDestBits, POINT3D const *pVertices,
        BITMAPINFO const *pTextureInfo,
        BYTE *pTextureBits )
{
    int Top, Middle, Bottom;
    int MiddleCompare, BottomCompare;
    float Y0 = pVertices[0].Y;
    float Y1 = pVertices[1].Y;
    float Y2 = pVertices[2].Y;

    // sort vertices in y
    if(Y0 < Y1) {
        if(Y2 < Y0) {
            Top = 2; Middle = 0; Bottom = 1;
            MiddleCompare = 0; BottomCompare = 1;
        } else {
            Top = 0;
            if(Y1 < Y2) {
                Middle = 1; Bottom = 2;
                MiddleCompare = 1; BottomCompare = 2;
            } else {
                Middle = 2; Bottom = 1;
                MiddleCompare = 2; BottomCompare = 1;
            }
        }
    } else {
        if(Y2 < Y1) {
            Top = 2; Middle = 1; Bottom = 0;
            MiddleCompare = 1; BottomCompare = 0;
        } else {
            Top = 1;
            if(Y0 < Y2) {
                Middle = 0; Bottom = 2;
                MiddleCompare = 3; BottomCompare = 2;
            } else {
                Middle = 2; Bottom = 0;
                MiddleCompare = 2; BottomCompare = 3;
            }
        }
    }

    gradients Gradients(pVertices);
    edge TopToBottom(Gradients,pVertices,Top,Bottom);
    edge TopToMiddle(Gradients,pVertices,Top,Middle);
    edge MiddleToBottom(Gradients,pVertices,Middle,Bottom);
    edge *pLeft, *pRight;
    int MiddleIsLeft;

    // the triangle is clockwise, so
    // if bottom > middle then middle is right
    if(BottomCompare > MiddleCompare) {
        MiddleIsLeft = 0;
        pLeft = &TopToBottom; pRight = &TopToMiddle;
    } else {
        MiddleIsLeft = 1;
        pLeft = &TopToMiddle; pRight = &TopToBottom;
    }

    int Height = TopToMiddle.Height;

    while(Height--) {
        DrawScanLine(pDestInfo,pDestBits,Gradients,
```

Listing 1. Continued from p. 22

```
    Height = YEnd - Y;

    float YPrestep = Y - pVertices[Top].Y;

    float RealHeight = pVertices[Bottom].Y - pVertices[Top].Y;
    float RealWidth = pVertices[Bottom].X - pVertices[Top].X;

    X = ((RealWidth * YPrestep)/RealHeight) + pVertices[Top].X;
    XStep = RealWidth/RealHeight;
    float XPrestep = X - pVertices[Top].X;

    OneOverZ = Gradients.aOneOverZ[Top] +
        YPrestep * Gradients.dOneOverZdY +
        XPrestep * Gradients.dOneOverZdX;
    OneOverZStep = XStep *
        Gradients.dOneOverZdX + Gradients.dOneOverZdY;

    UOverZ = Gradients.aUOverZ[Top] +
        YPrestep * Gradients.dUOverZdY +
        XPrestep * Gradients.dUOverZdX;
    UOverZStep = XStep *
        Gradients.dUOverZdX + Gradients.dUOverZdY;

    VOverZ = Gradients.aVOverZ[Top] +
        YPrestep * Gradients.dVOverZdY +
        XPrestep * Gradients.dVOverZdX;
    VOverZStep = XStep *
        Gradients.dVOverZdX + Gradients.dVOverZdY;
}

/********** DrawScanLine ************/

void DrawScanLine( BITMAPINFO const *pDestInfo,
        BYTE *pDestBits, gradients const &Gradients,
        edge *pLeft, edge *pRight,
        BITMAPINFO const *pTextureInfo,
        BYTE *pTextureBits )
{
    // we assume dest and texture are top-down

    int DestWidthBytes =
        (pDestInfo->bmiHeader.biWidth + 3) & ~3;
    int TextureWidthBytes =
        (pTextureInfo->bmiHeader.biWidth + 3) & ~3;

    int XStart = ceil(pLeft->X);
    float XPrestep = XStart - pLeft->X;

    pDestBits += pLeft->Y * DestWidthBytes + XStart;

    int Width = ceil(pRight->X) - XStart;

    float OneOverZ = pLeft->OneOverZ +
            XPrestep * Gradients.dOneOverZdX;
    float UOverZ = pLeft->UOverZ +
            XPrestep * Gradients.dUOverZdX;
    float VOverZ = pLeft->VOverZ +
            XPrestep * Gradients.dVOverZdX;

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

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

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

Programming Digitized Sound On the Sound Blaster

Figure 1. Comparing Analog and Digitized Waves



In the early days of PC sound, we had the PC speaker and its lesser-known sidekick, the Programmable Interval Timer (PIT). You could program the frequency of this hardware timer, and hook the timer up to the speaker. By changing this frequency, you could produce a surprising variety of sound effects and music.

There was only one problem—gamers and programmers hated it. So the world moved onto generation two: the AdLib FM Synthesizer. You could program it to produce decent music and even certain sound effects.

There was still a problem. The sound effects stunk.

Along came the Sound Blaster. It combined a digital-to-analog converter (DAC) with an FM synthesizer. Now, game programmers had access to decent music and decent sound effects. Finally.

There was much rejoicing. Until the question arose as to how to program the thing.

In developing DiamondWare's Sound ToolKit, we discovered scarce and poorly written documentation on the Sound Blaster. In this article, we'll give you the information you need to start developing your own digitized sound code.

Theory of Digitized Sound

As with most areas of computer programming, it is good to know some general background theory. It's often the case that the "easy" pitfalls are far from obvious. With sound, there are a number of possible problems that will cause audible glitches.

When digitized sounds are produced from an analog source, signals travel down an interconnect cable, go through a preamplifier, and are eventually processed by an analog to digital converter. We mention this because each step will have an effect on sound quality and overall volume.

Figure 1 shows the difference between continuous (analog) and discrete (digitized) waves. Unlike analog waves, digitized sounds are quantized, meaning that the amplitude at each point can have only one of a finite number of values. In addition, the sounds are discretely sampled; that is, the value is sampled periodically. Both facts raise important considerations. We'll discuss the amplitude problem first, then the time problem.

The range of values for each sample is finite. In a typical game playback system, this is 8 bits, or -128 to +127. The softest sound that can be represented is ±1, and any sound softer than this threshold will be lost at 0. The loudest sound is about ±127. Any sound louder than this is clipped, producing a harsh and nasty noise.

When you're mixing two or more sounds together, this clipping limit applies to the total mix. Thus, you want to produce your sounds at low volume levels so you can mix several of them without problems. If you are stuck with preproduced sounds, there are two ways to cut their dynamic range. The first is straightforward—you divide each sample by a constant. This is often implemented as a shift-right operation. It's fast and easy, but if you divide too much you will lose the softer parts of your sounds.

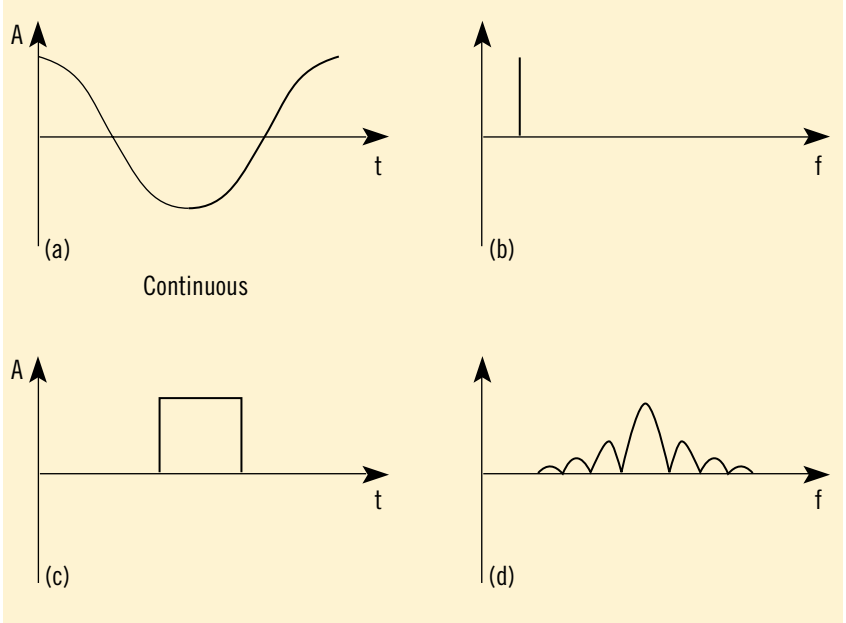
The other method is to dynamically compress your sounds. Basically, this will soften the louder parts and raise the softer parts. This is what pop radio stations do. After this process, the entire sound is equally loud. The implementation of this is beyond the scope of this article, but we mention it here for completeness. Quality sound processing software has the ability to compress dynamic range.

A digitized sound is considered a discretely sampled waveform because it is not contiguous in the time domain. In other words, you're sampling the waveform at T=1 and T=2 but there's no information for T=1.5. When it's played back, there's obviously sound at every moment—even in between the sample points, which is the crux of the problem.

To understand this dilemma, let's look at analog waveforms. It turns out that every possible analog waveform can be represented by the sum of a finite number of pure sine waves.

The plot of a sound on an oscillo-

Figure 2. Wave and Spectrum/Pulse and Spectrum



Keith Weiner
Erik Lorenzen

It's the low-down dirty details of making your game sing. And talk. And growl. And make funny little beeping sounds...it's... Sound Blaster!!!

scope is a time-domain plot, as shown in Figure 2a. That is, while the Y-axis represents amplitude, the X-axis represents time. It's possible to transform a wave into the frequency-domain. A plot after such a transformation, shown in Figure 2b, uses the X-axis to represent frequencies; each point farther to the right would be a higher frequency. You could read the graph to determine, for example, how loud the 1KHz component is. The highest frequency sine wave in this series is no higher than the highest frequency in the original wave. Figure 2c illustrates a square-wave pulse, comprising an infinite frequency of components, as you can see in its frequency-domain plot, shown in Figure 2d.

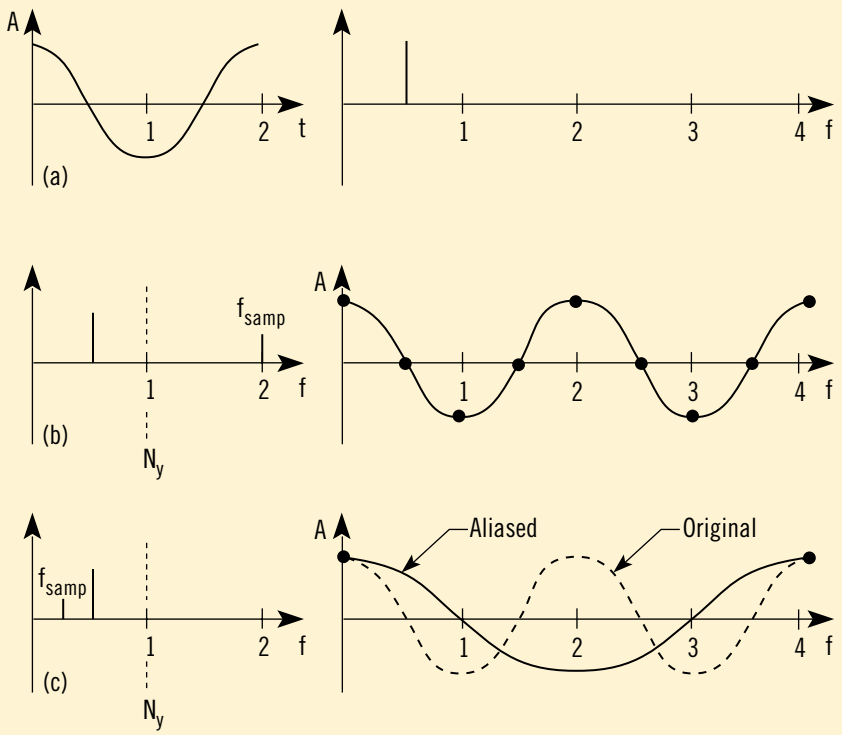
Now, let's get back to digital sound. It turns out that to "capture" a given frequency, you need to sample the wave at a rate at least twice as high as that frequency. For example, if you had a sound at 1KHz, you should sample it at 2KHz or higher. This is known as the Nyquist rate.

Figure 3 demonstrates the effects of sampling a wave above and below the Nyquist rate. In Figure 3a, we see the original analog wave and its spectrum in the frequency domain. If this signal is sampled above the Nyquist rate, as shown in Figure 3b, it can be correctly reconstructed. If the signal is sampled below the Nyquist rate, as shown in Figure 3c, there is not enough information to properly reconstruct the wave and aliasing occurs.

How can we capture a sine wave with only two points per cycle? We know in advance that it *is* a sine wave. There's only one possible sine wave that can be formulated, given two points during each cycle. Any waveform can be broken down into sine waves, so we have a way of discretely capturing (digitizing) analog sounds.

There's one last issue we must cover—playback. Because the waves are discretely sampled and are represented by few points, they're going to look quite square, as shown in Figure 4. Square looking digital signals connote "high frequency." If you play them back as is, they're going to have lots of noise—in fact, you'll hear a complete harmonic overtone of your sample. This lends a nasty metallic sound. The solution is to filter the output of the DAC in analog, eliminating all frequencies above half the sampling rate.

Figure 3. Sampling Rates and Aliasing



The Sound Blaster Family

We've seen 10 models of Sound Blaster from Creative Labs and dozens of clones from third parties. Fortunately, they all share the common architecture first presented in the Sound Blaster 1.5. By far, the Sound Blaster 16, Sound Blaster PRO2, and Sound Blaster 2.1 sold the most units, and they're still selling today.

This article will focus on the Sound Blaster 1.5, because it's the lowest common denominator. The higher models all support the Sound Blaster 1.5's modes.

Detecting the Sound Blaster

You can find the Sound Blaster by parsing the `BLASTER` environment variable. It's the easiest method, and it's recommended by Creative Labs and other manufacturers, especially because it can't crash the user's machine. This is how we'll do it here.

The `BLASTER` environment variable comprises several sections. A typical `BLASTER` variable might look like this:

```
BLASTER=A220 I5 D1 H5 P330 M250 T6
```

where:

- `A` is the base port address (here, it's `220h`). Values may be `210h` to `280h`.
- `I` is the IRQ (interrupt request) level. Values may be 2, 3, 5, 7, or 10.
- `D` is the 8-bit DMA channel. Values may be 0, 1, or 3.
- `H`, if present, is the 16-bit DMA channel. Values may be 5, 6, or 7.
- `P`, if present, is the port for `MPU401`, either `300h` or `330h`.
- `T` is the type of Sound Blaster card.

You can program the Sound Blaster in 14 easy steps:

- Reset the DSP (digital signal processor) and put it into a known state.
- Set up your interrupt service routine (ISR).
- Enable the IRQ the Sound Blaster card is using.
- Program the DAC speaker.
- Program the DMA controller for a single cycle transfer.

- Program the playback rate (time constant).
- Program the DSP output/input for single-cycle transfer.

Transfer begins immediately after step 7. At this point, you can go draw some graphics, read a disk, and get some coffee. When the buffer is done, the Sound Blaster will generate an interrupt. This will transfer control to the ISR.

Next:

- Acknowledge the DSP.
- Send the programmable interrupt controller (PIC) an end of interrupt (EOI).
- Program the Sound Blaster to play another buffer or set a flag to show that we are done playing.

After we have finished all data transfers, we need to:

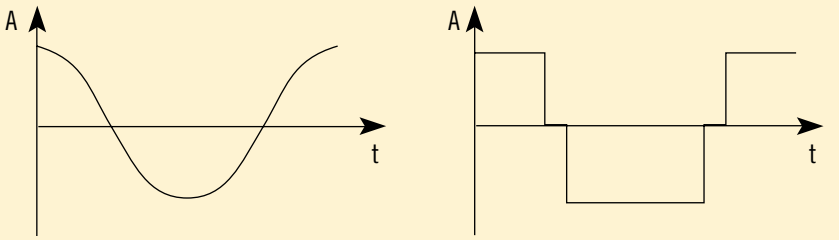
- Disable the DAC speaker.
- Disable the IRQ.
- Unhook the ISR.
- Reset the Sound Blaster DSP, leaving it in a good state, ready to work with other applications.

Detect and Reset

Before we assume that the presence of the BLASTER variable means we have a Sound Blaster in the system, we can check to see if a DSP really does exist at the specified port, shown in Table 1, and attempt to send it a reset by doing the following:

- Write a 1 to the sb_RESET port.
- Wait three microseconds (μsec).
- Write a 0 to sb_RESET.
- Read sb_READ_STATUS (up to 65,535 times), waiting for the msb (most significant bit) to be set.
- If the msb never gets set, no Sound Blaster card is present.
- Read sb_READ_DATA. If the return value is AAh, a Sound Blaster card is present.
- If the return value is not AAh, repeat Steps 4 to 6 until the count runs out, or a Sound Blaster card is found.

Figure 4. Comparing a Digitized Wave to the Original Analog Wave



Original wave | Digitized wave

Reading and Writing

To read data from the DSP, we must read from sb_READ_STATUS until the msb is set. Then, read from sb_READ_DATA:

```
unsigned sb_ReadDSP(unsigned baseport)
 while(!(0x80 & inp(baseport +
  sb_READ_STATUS))); //waiting for the
                     //MSB to be set
 return((unsigned)inp(baseport +
  sb_READ_DATA));
}
```

To write to the DSP, read from sb_COMMAND_STATUS until the msb is reset. Then, write the desired command (or command data) to sb_WRITE_COMMAND:

```
void sb_WriteDSP(unsigned baseport,
 unsigned value) {
  while(0x80 & inp
  (baseport +sb_WRITE_STATUS));
  // wait for the MSB to be clear
}
  outp(baseport + sb_WRITE_COMMAND,
   (int)value);
}
```

Handling DSP Interrupts

The DSP will generate an interrupt when ever it's done recording or playing a DMA buffer. To keep the system from crashing and the Sound Blaster playing, we need to set up an ISR. Each interrupt must be acknowledged by reading sb_ACKIRQ. This tells the DSP that you have received the interrupt, and it can stop pulling the line.

Using DSP commands

A DSP revision of 1.xx accepts 20 commands, as shown in Table 2. We will only need to use five commands to get up and running. To simplify the following discussion, we will use the example functions sb_ReadDSP and sb_WriteDSP.

The DAC speaker controls what we hear (and what the Sound Blaster hears). With the speaker on, we can hear the digitized playback, but the Sound Blaster can't hear us (record), and vice versa.

To turn on the speaker, send sb_DACSPKRON to the DSP and wait 112 μsec for the DSP to complete the operation:

```
sb_WriteDSP(baseport, sb_DACSPKRON);
```

Turning the speaker off is similar; send sb_DACSPKROFF to the DSP and wait even longer (220 μsec)—no one said this hardware was fast:

```
sb_WriteDSP(baseport, sb_DACSPKROFF);
```

The sb_SETTIMECONST command sets

Table 1. Sound Blaster DSP Ports

Port Number	Write	Read	During IRQ
2X6h	sb_RESET	None	Normal
2XAh	None	sb_READ_DATA	Normal
2XCh	sb_WRITE_COMMAND	sb_WRITE_STATUS	Normal
2XEh	None	sb_READ_STATUS	sb_ACKIRQ (read-only)

X denotes base address. For base address of 220h, the ports are 226h, 22Ah, 22Ch, and 22Eh.

Table 2. DSP.1 xx Commands

Number	Description
10h	Set polled output mode and PCM (uncompressed) samples.
14h	Set DMA output mode and PCM samples.
74h	Set DMA output mode and 8- to 4-bit ADPCM samples.
75h	Set DMA output mode and 8- to 4-bit ADPCM samples (with reference byte).
76h	Set DMA output mode and 8- to 3-bit ADPCM samples.
77h	Set DMA output mode and 8- to 3-bit ADPCM samples (with reference byte).
16h	Set DMA output mode and 8- to 2-bit ADPCM samples.
17h	Set DMA output mode and 8- to 2-bit ADPCM samples (with reference byte).
38h	Set polled output mode and MIDI data.
20h	Set polled input ode and PCM (uncompressed) samples.
24h	Set DMA input mode and PCM (uncompressed) samples.
30h	Set polled input mode and MIDI data.
31h	Set interrupt input mode and MIDI data.
D0h	Pause DMA.
D4h	Resume DMA.
40h	Set time constant
80h	Pause DMA for a specified duration.
E1h	Get DSP version.
D1h	Enable DAC speaker.
D3h	Disable DAC speaker.

how many samples per second the DSP will record or playback, but it doesn't take a sampling rate directly. We must convert from Hz to the Sound Blaster time constant. The time constant is always an unsigned byte:

```
tc = 256 - (1000000 / (num_channels *
 sampling_rate));
sb_WriteDSP(baseport, sb_SETTIMECONST);
sb_WriteDSP(baseport, rate);
```

To play a sound, send the DSP one of the output sound commands. We'll use `sb_PLAY8BITMONO`. Follow this command with 2 bytes representing the size of the buffer. The buffer can be between 1 and 65,536 bytes. No one would want to program the Sound Blaster to transfer 0 bytes, so 0 means 1 byte, and 65,535 means 65,536 bytes:

```
lowbyte = (unsigned char)(buffsize - 1);
highbyte = (unsigned char)((buffsize - 1)
 >> 8));
sb_WriteDSP(baseport, sb_PLAY8BITMONO);
sb_WriteDSP(baseport, lowbyte);
sb_WriteDSP(baseport, highbyte);
```

Interrupt Programming

No discussion of Sound Blaster DSP programming would be complete without mention of the 8259A PIC. Integrally related is the 80x86 processor's interrupt mechanism, including the vector table. Let's go over the steps involved in an IRQ and its handling.

The Sound Blaster must go through nine steps to build an IRQ:

- The Sound Blaster DSP signals the PIC that it wants to interrupt the CPU.
- The PIC checks the interrupt mask register (IMR) to see if this is cool.
- If so, the PIC checks to see if any higher-priority IRQ's are being serviced.
- If so, it waits until they send an end of interrupt (EOI) to the PIC.
- If not, the PIC sends a signal to the CPU over a dedicated line.
- If the interrupt flag is set, the CPU replies with an interrupt acknowledge (INTA).
- The PIC then sends an IRQ and the IRQ level.
- The CPU pushes the flags, CS, and IP registers—in that order—on the current stack
- The CPU jumps to the address specified in the vector table for this IRQ.

To respond to an IRQ (for ISR's only):

- Tell the hardware (the Sound Blaster DSP) to stop pulling the interrupt line.
- Send an EOI to the PIC.
- Set a global variable—a flag—for main program loop (this step is optional).
- Prepare the next sound buffer.
- Return from interrupt (IRET instruction).

There's a simple `INT 21` (DOS) call to hook the interrupt vector, which is even easier in C. We also need to enable our interrupt in the PIC itself. To do this, read the IMR, reset the bit corresponding to the IRQ level to which the Sound Blaster DSP is set, and write the IMR:

```
#define dig_IMRPORT 0x21
temp = inp(dig_IMRPORT); //Read the IMR
temp &= dig_onmask[irqlevel];
//Enableour channel
outp(dig_IMRPORT, temp); //Write the IMR
```

Programming the DMA Controller

The 8237A high-performance programmable direct memory access (DMA) controller provides a way to transfer data between memory and the I/O bus without using CPU. If you program it properly, it allows for easy and nearly overhead-free data transfers. Make a mistake, however, and you've as good as sent a garbage truck to dump a pile of trash in memory!

An AT-class machine has two DMA controllers and eight DMA channels. The DMA controllers have 44 I/O ports and four modes of operation.

We'll only discuss channels 0, 1, and 3 (2 is used by the floppy controller). These are the 8-bit channels. Channels 4 to 7 are 16-bit channels and aren't used by 8-bit Sound Blasters.

We're interested in single-cycle DMA mode, which means one byte is transferred by the DMA controller for each data request (DREQ) it receives from the Sound Blaster DSP.

There are nine steps to programming a DMA controller. Steps 2 through 8 employ either shared registers—used for

all channels—or channel specific registers:

- Disable interrupts.
- Disable our DMA channel (shared register).
- Reset the flip-flop (shared register).
- Set our channel's mode (channel-specific register).
- Program the address register (channel-specific register).
- Program the page register (channel-specific register).
- Program the count register with one less than the actual transfer count (channel-specific register).
- Enable the DMA channel (shared register).
- Enable interrupts

The DMA controller works with physical addresses, not with `segment:offset` addresses, not with selectors, and so on. In real mode, it's easy to translate from `segment:offset` to a physical address (protected-mode selectors can be translated as well). The DMA controller works with a page number, which is literally the physical address divided by 65,536. A DMA buffer cannot cross such a physical page address; you must verify that your buffer meets this criterion! Within the physical page, the DMA controller increments an offset. Code to translate from `segment:offset` to physical page and offset is:

```
off = (*((unsigned _far *)&(sound)));
seg = (*((unsigned _far *)&(sound) + 1));
seg <<= 4;
padd = seg + off;   // calc physical
                    address
page = padd >> 16;  // calc page number
```

Don't be put off by our method of obtaining the segment and offset of a pointer. This may seem complex, but we're simply taking a pointer to sound; the first word this points to is the offset, and the second is the segment.

Explanations of the workings of the DMA controller tend to be very lengthy. Fortunately, the DMA controller is very well documented. We refer you to two books that provide in-depth explanations for further reading: *Developer Kit for Sound Blaster Series, 2nd Ed.* (Creative Labs, 1993); and *The Indispensable PC Hardware Book* (Addison-Wesley, 1993), by Hans-Peter Messmer.

The Code

The code that accompanies this article compiles with Microsoft C/C++ 7, Borland C/C++ 3.1 and 4.0. It should port easily to other DOS C environments. We used the large memory model when we compiled it. We tested it with a Sound Blaster 1.5, 2.1, Pro 2, 16, and AWE32. The code is available on CompuServe in the SDFORUM in the GDMag Library. ■

Keith Weiner and Erik Lorenzen have developed a sound toolkit for DOS games, called DiamondWare's Sound ToolKit (available from MVP Software). Contact them via e-mail at keith@dw.com or erik@dw.com.

THE GLOSSARY

Working with the Sound Blaster can require a graduate degree in the language of acronyms. Hence, we provide here a breakdown of the abbreviations used throughout this article.

EOI	**End of interrupt**
DAC	**Digital-to-analog converter**
DMA	**Direct memory access**
DREQ	**Data request**
DSP	**Digital signal processor**
IMR	**Interrupt mask register**
INTA	**Interrupt acknowledge**
IRQ	**Interrupt request**
ISR	**Interrupt service routine**
PIC	**Programmable interrupt controller**
PIT	**Programmable interval timer**

Supercharge Your Sprites

As we continue on our quest for the fastest graphics performance possible, we are going to focus on drawing sprites. Just what is a sprite? A sprite is usually defined as a graphical image that moves independently on the screen. Characters in video games are sprites. So are bullets, explosions—even the cursor is a type of sprite.

Unlike tiles or text, sprites often come in irregular shapes and sizes. These varied shapes often make it difficult to optimize a general-purpose drawing routine. To achieve superior performance when drawing sprites, we are going to examine a technique called "compiled sprites," sometimes referred to as "compiled bitmaps."

Normal Sprites

To start, we need to understand how sprites are usually implemented. Today, most sprites are drawn by a programmer or artist using a paint program or some form of sprite editor, which itself is a custom paint program. The program draws the image in a rectangular work area, using a palette of colors.

When you begin a new sprite, the work area is filled with a color, usually black, white, or grey, that you've designated as a "transparent" or background color. The actual color doesn't matter—it's simply a value that the programmer decides will be used to indicate transparency.

When the sprite is drawn by the program, only the portions of the image block that are *not* the transparent color are actually drawn. The areas with the transparent color are left alone, and the background shows through.

Because irregular shapes are so difficult to code for, you would usually choose to store a sprite in memory as a rectangular array of pixels. A normal sprite drawing routine then goes through the following steps. First, it computes the screen address that corresponds to the upper left pixel in the sprite's array, even if the pixel is transparent. Then it scans the sprite's data from left to right and top to bottom, checking each pixel to see if it is transparent. If it is not, the program draws that pixel to the screen at an offset corresponding to that pixel's position in the sprite's data array.

This method works very well and you can optimize it with assembly language. But it is not as fast as routines that draw solid blocks or tiles, like the ones I presented in "Faster Image Drawing" (Feb./Mar. 1995). And the worst part is that it is slower than these routines despite the fact that fewer pixels are actually drawn. This wouldn't be a big deal if we didn't use sprites very much, but for many action-oriented games, we use them extensively.

Imagine you are writing a Super Galaxian-style game. The player's ship, shields, and every bullet, missile, explosion, and enemy ship is an irregularly shaped sprite. You will have to redraw every object for each frame of gameplay. If your sprite drawing routines aren't very good, the game is going to bog down when the action gets fast and furious. Now it becomes obvious why you want the fastest sprites possible.

Performance is lost when you draw

Matt Pritchard

Sprites come in all shapes, sizes, and colors. This variance often presents performance challenges for developers. Here are some tips to help you achieve superior performance when drawing sprites.

irregularly shaped sprites not because the code is inefficient, but because for every pixel, a decision must be made whether or not it should be actually drawn. In sprite-drawing routines, it turns out that not drawing a pixel usually imposes a big performance hit on normally tight assembly language code. This happens because the CPU has to jump, flush its prefetch queue, and reload it. The code could be reversed, but the performance would suffer every time a pixel is drawn. You must take one of two paths because there is no way around the fact that a decision must be made. Or is there?

Looking Back, Looking Forward

It the earliest days of 8-bit computers, a game would often have dedicated code to draw individual sprites. Some systems even had hardware support for sprite graphics. (Here's a historical trivia fact: The Atari 2600 game system only had 128 bytes of RAM, so each line of each sprite had to be drawn with custom code in ROM.)

Programmers would hand-code routines to draw each line, box, and pixel that made up a sprite. The advantage was that only the pixels that needed to be drawn were coded. The biggest drawback was that it was a slow process, requiring the programmer to change code every time one single bit in a sprite was changed, possibly introducing bugs each time. As graphics got bigger and changed more frequently during development, this method proved less practical and general purpose routines took over.

But what if the computer could automatically generate code that draws each sprite, knowing exactly what to draw and what to eliminate for each image? All the comparisons, branching, and other decision-making code that would normally be executed every time the sprite was drawn or moved would be eliminated. The processing time saved for every sprite in every frame drawn would add up quickly. That is the basic premise behind compiled sprites.

Compiled Sprites

There are two main categories of compiled sprites. One is what I call "data compiled," which converts the sprite image into a stream of data that is fed to a special routine. All the calculations, comparisons, and decisions are made by the compiling routine during the compiling process and encoded into the data stream.

The drawing routine itself is a simple state machine that reads each chunk of data and processes it in exactly the same way. The inner loops are unrolled and contain no comparisons or branches to slow things down. The data stream provides all the information needed. This technique provides a significant performance increase and is often used with larger or compressed graphic images.

The second category consists of what I call "code compiled" sprites. The program generates actual machine language instructions that draw the specific image on the screen. There are no decision or control instructions, only direct writes to video memory. This method provides additional performance gains

Listing 1. A 256-Color Sprite (Continued. on p. 38)

```
/*  ========================================================================   */
/*  COMPLBMP.C - Routine to compile a 256-color bitmap image for               */
/*                  Mode X or Mode 13h.                                        */
/*  Author:  Matt Pritchard for Game Developer Magazine.                       */
/*           Adapted from MODEX108                                             */
/*  ========================================================================   */

    /*  This stuff could go into a .h file           */

    /*  Macro Definitions needed by Compile_Bitmap    */

#define ucharf   unsigned char far
#define uchar    unsigned char
#define uint     unsigned int

#define hi_word( x ) (unsigned char)  (x >> 8)
#define lo_word( x ) (unsigned char)  (x & 0x00FF)

    /* Prototypes for Compiled Bitmap Routines */

ucharf * Compile_Bitmap (ucharf *, int, int, int, int);

void far pascal draw_compiled_bitmap (uchar far *, int, ints);
void far pascal draw_compiled_bitmap_13h (uchar far *, int, ints);

/* ---------------------------------------------------------------------------*/

/*  This function takes a Sprite that is stored in a two-dimensional array, such
    as char ImageData[32][32], and creates a buffer that contains the machine
    language code to quickly draw that image in Mode 13h or Mode X.

    The sprite data is stored line by line, from top to bottom.  Each line
    is stored from left to right.  A transparant color value is used to
    indicate which pixels are not part of the image and should not be drawn.

    Because Mode X supports various screen sizes, we must know the width
    of the screen a sprite will be displayed on in advance.  For Mode 13h,
    that width is normally 320.

    When possible, two adjacent pixels will be drawn with one 16-bit MOV
    instruction.  This results in smaller and faster code.

    This function allocates a buffer to hold the compiled code and
    returns a far pointer to it.  The pointer need only be a char type
    pointer, since our assembly language routine does the actual calling of it.

    If the sprite is too big or the program has run out of memory, a null pointer
    is returned, otherwise a pointer to the compiled code is returned.
*/

ucharf * Compile_Bitmap (ucharf * theImage,   /* Far Ptr to the Sprite */
int X_width,                                  /* Width of the Sprite   */
int Y_width,                                  /* Height of the Sprite  */
int Trans_Color,                              /* Transparent Color     */
int Screen_Width)                             /* Width of the screen   */
{

    int x, y, p;                              /* Loop counters for X, Y, and plane  */
    int Words, Bytes;                         /* Count of each type of instruction  */
    int b1, b2;                               /* Valid pixel flags                  */
    uintVidOffset, Offset;                    /* Offsets for memory calculations    */

    int BytesPerLine;                         /* Width of display in address bytes  */
    longCompiledBufferSize;                   /* The size of the compiled sprite code*/
    longc;                                    /* Counter for the code writing loop  */

    ucharf * theBuffer;                       /* Pointer to the compiled code buffe */

    int Num_Planes;                           /* The number of video planes (4 or 1)*/
    int Next_Pixel;                           /* The number of bytes between adjacent pixel*/
    int Code_Overhead;                        /* Size of any overhead code needed      */

/*  The variable Mode_X controls if we are compiling a sprite for Mode 13h or
    Mode X.  For Mode X, we must split the image into four separate planes and
```

over data compiled sprites, and is what we will now focus on. From this point on, when I refer to compiled sprites, I'll be talking about code-compiled sprites.

For this article, I have written a routine in C, shown in Listing 1, which will compile a 256-color sprite for either Mode X or Mode 13h. I've also provided the assembly language setup routines needed to call the compiled sprite code in Listing 2. With minor modifications, you should be able to use these routines in any program you wish. But I'll bet you aren't satisfied with just using the routines; you want to know what makes them tick. Let's take a closer look at the routines and discuss how they work.

> The program generates actual machine language instructions that draw the specific image on the screen.

How it Works in a Program

First, here's a quick overview of where the routines fit into a typical game. Normally, sprites are loaded from disk and placed into memory buffers either at startup or at some stopping point, such as a between rounds screen. With the sprite image in memory, the program

Listing 1. A 256-Color Sprite (Continued from p. 36)

```
    add plane switching code to the compiled sprite.  If the value of Mode_X is
    0, we compile for Mode 13h, otherwise we compile for Mode X.                */

    int Mode_X = -1;     /* -1 = Mode X,  0 = Mode 13h                         */

    if (Mode_X) {
        Num_Planes = 4;
        Next_Pixel = 4;
        Code_Overhead = 20;
    } else {
        Num_Planes = 1;
        Next_Pixel = 1;
        Code_Overhead = 5;
    }

    BytesPerLine = Screen_Width / Num_Planes;

/*  First, we pass through the bitmap and count up the number of adjacent pixel
    pairs and the number of single pixels. With this information, we will know
    how big a buffer to allocate.                                               */
    Words = Bytes = 0;

    for (p = 0; p < Num_Planes; p++)
    {
        for (y = 0; y < Y_width; y++)
        {
            Offset = y * X_width;
            for (x = p; x < X_width; x+=Next_Pixel)
            {
                /* Check the current pixel to see if it should be displayed    */

                b1 = (theImage[Offset+x] != Trans_Color) ? -1 : 0;

                /*  Check the next adjacent pixel (if there is one), and see if
                    it should also be displayed                                 */

                if  ((x + Next_Pixel) < X_width) {
                    b2 = (theImage[Offset+x+Next_Pixel] != Trans_Color) ? -1 : 0;
                } else {
                    b2 = 0;
                }

                /* Check for a pair of adjacent pixels, or a lone single pixel */

                if (b1) {
                    if (b2) {
                        Words++;    /*  Another adjacent pixel pair            */
                    x+=Next_Pixel;  /*  Skip over the next pixel               */
                        } else {
                    Bytes++;        /*  One more lone pixel                    */
                    }
                }
            }
        }
    }

/*  Determine how big a buffer we need for the compiled code, allocate it, and
    get a far pointer to it. */
    CompiledBufferSize = Code_Overhead + (6 * Words) + (5 * Bytes);

/*  Here is where the users can insert their own error handling code            */
    if (CompiledBufferSize > 65535) {
        /*  Error; compiled sprite would be too large (greater than 64K).       */
        return (0);
    }

    if ( (theBuffer = (ucharf *) malloc( (size_t) CompiledBufferSize)) == 0) {
        /*  Error allocating buffer; out of memory.                             */
        return (0);
    }

/*  Now, we go through the image again, this time creating the code to write into
    the compile code buffer.                                                    */
```

calls the `compile_sprite` routine with basic information about the sprite.

The `compile_sprite` function allocates a second memory buffer and fills it with machine language instructions made from the sprite data. Then, it returns a pointer to the newly compiled code to the program. Depending on what is needed by your program, you can deallocate the original sprite buffer, freeing up additional memory. This compiling process isn't necessary each time the program is run; the code produced is completely relocatable, so you could load and save the compiled sprites just like regular sprites.

When the program needs to draw a sprite, it calls a special routine with a pointer to the compiled sprite code and the X,Y screen position at which the program will draw the sprite. This routine first sets up the CPU's registers with the correct position on the screen and then directly executes the compiled sprite code by jumping to it. The compiled sprite code then actually returns to the routine that called the special routine. To the programmer, this process closely resembles calling a normal sprite drawing routine.

How it Compiles the Sprite

When we call the routine to compile the sprite, we send it the following information about the sprite:

- A pointer to an array containing the sprite image
- The width of the array in bytes
- The height of the array in lines
- The color value that will indicate transparent pixels
- The width of the screen at the time the sprite will be drawn.

The routine needs that last item, width of the screen, to compute the offsets into display memory for each pixel. In Mode 13h, the width will always be 320, but in Mode X (see my article "Mode X Revealed," Dec. 1994), it can vary according to the programmer's wishes.

Compiling the sprite is a two-pass process. On the first pass, the sprite image is scanned, and each pixel is examined to see if it is a transparant

pixel. Every normal pixel is checked to see if it is adjacent to another normal pixel. This process generates a count of how many single pixels and adjacent pixel pairs make up the sprite.

Once the pixels are counted, the routine calculates the size of the compiled sprite code, and allocates memory to hold the code. Speaking of memory, how much memory do compiled sprites take up? The size of the compiled sprite buffer will vary from image to image, but will be approximately 4 bytes per pixel drawn. Transparent areas produce no code. If a sprite is stored in a 32-by-32-pixel grid, but only about half the pixels are actually drawn, the grid will take up 1K, while the compiled sprite should take up about 2K.

The size varies because our routine creates code to draw two adjacent pixels with one instruction whenever possible. The actual code created is:

`MOV [BX+offset], <16-bit constant>,`

which is 6 bytes long and draws two pixels. For pixels that are stuck by themselves, the code created is:

`MOV [BX+offset], <8-bit constant>,`

which is 5 bytes long and draws a single pixel.

The routine makes a second pass through the image, this time generating actual code whenever a single pixel or pixel pair is encountered. If we are compiling for Mode X, plane switching code is generated in between each video plane. Finally, instructions to return to the calling routine are added at the end of the buffer.

Mode 13h vs. Mode X

For Mode 13h, compiled code creation is very straightforward. But for Mode X, things get more complicated. In Mode X, we have to draw all the pixels on each video plane before going to the next video plane. Because of the video planes in Mode X, the two adjacent pixels we compile into one instruction are actually four pixels apart on the screen. In addition, we must include a 5-byte sequence to select the next video plane into the

Listing 1. Continued from p. 38

```
    c = 0;
    for (p = 0; p < Num_Planes ; p++)
    {
        for (y = 0; y < Y_width; y++)
        {
            Offset = y * X_width;
            for (x = p; x < X_width; x+=Next_Pixel)
            {

            /*  Check the current pixel to see if it should be displayed      */

            b1 = (theImage[Offset+x] != Trans_Color) ? -1 : 0;

            /*  Check the next adjacent pixel (if there is one), and see if it
                should also be displayed                                      */

            if ((x + Next_Pixel) < X_width) {
                b2 = (theImage[Offset+x+Next_Pixel] != Trans_Color) ? -1 : 0;
            } else {
                b2 = 0;
            }

            /*  Generate code for a pair of pixels, or for a single pixel.    */

            if (b1) {
                VidOffset = (BytesPerLine * y) + ((x-p) / Num_Planes);

        if (b2) {           /*  Create Code to write Word Constant           */

            theBuffer[c++] = 0xC7;                      /*  MOV word ptr     */
            theBuffer[c++] = 0x87;
            theBuffer[c++] = lo_word( VidOffset );      /*  BX+VidOffset     */
            theBuffer[c++] = hi_word( VidOffset );
            theBuffer[c++] = (uchar) theImage[Offset+x];
            theBuffer[c++] = (uchar) theImage[Offset+x+Next_Pixel];

            x+=Next_Pixel;                              /*  Skip over second pixel*/

        } else {                                        /*  Create Code to write Byte Constant*/

            theBuffer[c++] = 0xC6;                      /*  MOV byte ptr     */
            theBuffer[c++] = 0x87;
            theBuffer[c++] = lo_word( VidOffset );      /*  BX+VidOffset     */
            theBuffer[c++] = hi_word( VidOffset );
            theBuffer[c++] = (uchar) theImage[Offset+x];
                }
            }
        }

        if ( (Mode_X) && (p < 3) ) {                    /*  Generate plane switching code*/

            theBuffer[c++] = 0xD0;                      /*  ROL AL, 1   ; Get New mask*/
            theBuffer[c++] = 0xC0;
            theBuffer[c++] = 0x13;                      /*  ADC BX, CX  ; Add in Addr wrap*/
            theBuffer[c++] = 0xD9;
            theBuffer[c++] = 0xEE;                      /*  OUT DX, AL  ; Select new Plane*/
        }
    }

    /*  Create exit code to return to the calling program */

    theBuffer[c++] = 0x5D;                              /*  POP BP  ; Restore BP     */
    theBuffer[c++] = 0x1F;                              /*  POP DS  ; Restore DS     */
    theBuffer[c++] = 0xCA;                              /*  RETF8   ; Exit & Clean Up Stack*/
    theBuffer[c++] = 0x08;
    theBuffer[c++] = 0x00;

    /*  Return a pointer to the Buffer containing the Compiled Code           */

    return (theBuffer);

}
```

Listing 2. Compiled Sprite Setup and Call Routine (Continued on p. 41)

```
; =======================================================;
; COMPLBMP.ASM  - Compiled Sprite Setup & Call Routines for;
;                 Mode X or Mode 13h.                     ;
; Author;  Matt Pritchard for Game Developer Magazine.    ;
;          Adapted from MODEX108                          ;
; Assembler Used; MASM 5.10a                              ;
; =======================================================;

.MODEL Medium
.286
.CODE

; ===== General Constants & Macros =====

wp  EQU WORD PTR
dp  EQU DWORD PTR
fp  EQU FAR PTR

; ===== VGA Register Values & Constants =====

VGA_Segment       EQU 0A000h            ;Vga Memory Segment

SC_Index          EQU 03C4h             ; VGA Sequencer Controller
SC_Data           EQU 03C5h             ; VGA Sequencer Data Port

MAP_MASK_PLANE2   EQU 01102h            ; Map Register + Plane 1
PLANE_BITS        EQU    03h            ; Bits 0-1 of Xpos = Plane#

;=======================================================
;  DRAW_COMPILED_BITMAP (CompiledImage, X_pos, Y_Pos)
;=======================================================
;
; Sets up a call to a compiled bitmap in Mode X.
;
; ENTRY;   Image  = Far Pointer to Compiled Bitmap Data
;          Xpos   = X position to Place Upper Left pixel at
;          Ypos   = Y position to Place Upper Left pixel at
;
; EXIT;  No meaningful values returned
;

  DCB_STACK    STRUC
               DW  ?,?                  ; DS, BP
               DD  ?                    ; Caller
  DCB_Ypos     DW  ?                    ; Y position to Draw Bitmap at
  DCB_Xpos     DW  ?                    ; X position to Draw Bitmap at
  DCB_Image    DD  ?                    ; Far Pointer to Graphics Bitmap
  DCB_STACK    ENDS

  PUBLIC   DRAW_COMPILED_BITMAP

  DRAW_COMPILED_BITMAP    PROC    FAR

  Push DS                               ; Save DS
  Push BP                               ; AX-DX are destroyed
  Mov  BP, SP                           ; Set up Stack Frame

; Get DS;BX to point to (Xpos,Ypos) on the current
; display page in VGA memory

; ***** USER NOTE ***** MODIFY AS NEEDED *****
;
; Line_Offset is lookup table containing the start
; offset for each line in VGA display memory.
; Here, I assume it to be a table of word values
; which are stored in the current code segment.

  Mov  BX, [BP].DCB_Ypos            ; BX = Ypos
  Add  BX, BX                       ; Scale BX to Word Offset
  Mov  BX, wp CS;Line_Offset[BX]    ; Get Offset of Line Ypos

  Mov  AX, [BP].DCB_Xpos            ; Get UL Corner Xpos
  Mov  CL, AL                       ; Save Plane # in CL
  Shr  AX, 2                        ; X/4 = Offset Into Line
; ***** USER NOTE ***** MODIFY AS NEEDED *****
;
; CURRENT PAGE is a DWORD pointer to the currently active
; Mode X video memory page.  The first word is the offset
; into the video adaptor, and the second is the constant
; value of A000 - the VGA's graphics memory segment.
; Here, I assume it to be in DGROUP.

  Lds  DX, dp CURRENT_PAGE          ; Get Current VGA Page
  Add  BX, DX                       ; DS;BX->Start of Line
  Add  BX, AX                       ; DS;BX->Upper Left Pixel

; Select the first video plane, and set up the registers
; so the next 3 planes can be quickly selected.

  And  CL, PLANE_BITS               ; CL = Starting Plane #
  Mov  AX, MAP_MASK_PLANE2          ; Mask & Plane Select
  Shl  AH, CL                       ; Select correct Plane
  Mov  DX, SC_Index                 ; VGA Sequencer ports
  Out  DX, AX                       ; Set Initial Vid Plane
  Inc  DX                           ; Point DX to SC_Data
  Mov  AL, AH                       ; Mask for future OUT's
  Clr  CX                           ; CX = Constant 0

; Setup DS;    BX = Upper left corner of Image in VGA memory
;              BP = Local Stack Frame
;              AL = OUT mask for Selecting video Plane
;              CX = Constant value 0 for ADC
;              DX = SC_Data; VGA Sequencer Data Port
;              AH = Destroyed
;              SI,DI = Not modified during call
;
; Now we jump to the compiled code which actually draws the
; sprite.  The compiled code will return to the caller.

 Jmp   dp [BP].DCB_Image            ; Draw Sprite

  DRAW_COMPILED_BITMAP    ENDP



;=======================================================
;  DRAW_COMPILED_BITMAP_13h (CompiledImage, X_pos, Y_Pos)
;=======================================================
;
; Sets up a call to a compiled bitmap in Mode 13h.
;
; ENTRY;   Image  = Far Pointer to Compiled Bitmap Data
;          Xpos   = X position to Place Upper Left pixel at
;          Ypos   = Y position to Place Upper Left pixel at
;
; EXIT;  No meaningful values returned
;
```

compiled sprite buffer before we compile the pixels in the next video plane.

In Listing 1, the variable `Mode_X` controls the output of compiled sprite code. If it is nonzero, the image is broken into four memory planes, and the plane switching code is added to the compiled sprite. If `Mode_X` is zero, then code for `Mode_13h` is produced.

Listing 2 shows two separate functions for drawing a compiled sprite—one for Mode X and one for Mode 13h. The Mode X routine, `draw_compiled_bitmap`, loads registers with the screen address and plane-switching values. The Mode 13h routine, `draw_compiled_bitmap_13h`, only needs to load registers with the screen address.

Because compiling sprites reduces the process to its bare essentials, some limitations exist. You can't resize, rotate, or clip a compiled sprite to a rectangle. With normal sprites, you could write a different sprite drawing routine that performed these operations.

Suggestions for Improvements

The sprite compiling code I've presented here is only a start. There is no reason we can't include other capabilites in the compiled code. Clipping on one axis could be accomlished by ordering the compiled instrctions along that axis and using a jump table. Among the things that could be added are:

- Add support for EGA and VGA 16-color sprites
- Add support for SVGA and VESA modes
- Add support for word-aligned writes to video memory
- Add support for 32-bit instructions
- Add support for vertical or horizontal clipping.

I leave it to you, our reader, to decide where to take it. Until next time, happy coding! ■

Matt Pritchard is a software developer for Lacerte Software in Dallas, Texas, and the author of MODEX110, a comprehensive freeware ModeX library. You can reach him via e-mail at matthewp@netcom.com or through Game Developer *magazine.*

Listing 2. Compiled Sprite Setup (Continued from p. 40)

```
  PUBLIC    DRAW_COMPILED_BITMAP_13H

  DRAW_COMPILED_BITMAP_13H   PROC   FAR

  Push DS                                ; Save DS
  Push BP                                ; AX-DX are destroyed
  Mov  BP,,   SP                         ; Set up Stack Frame

; Get DS;BX to point to (Xpos, Ypos) in VGA memory

; ***** USER NOTE ***** MODIFY AS NEEDED *****
;
; Line_Offset is lookup table containing the start
; offset for each line in VGA display memory.
; Here, I assume it to be a table of word values
; which are stored in the current code segment.

  Mov  BX, [BP].DCB_Ypos                 ; BX = Ypos
  Add  BX, BX                            ; Scale BX to Word Offset
  Mov  BX, wp CS;Line_Offset[BX]         ; Get Offset of Line Ypos
  Add  BX, BP].DCB_Xpos                  ; Get UL Corner of Sprite

  Mov  AX, VGA_Segment                   ; Segment A000
  Mov  DS, AX                            ; DS;BX -> VGA memory

; Setup DS;    BX = Upper left corner of Image in VGA memory
;              BP = Local Stack Frame
;              AX = Destroyed
;       SI,    DI = Not modified during call
;       CX,    DX = Not modified during call
;
; Now we jump to the compiled code which actually draws the
; sprite.  The compiled code will return to the caller.

  Jmp  dp [BP].   DCB_Image              ; Draw Sprite

DRAW_COMPILED_BITMAP_13H  ENDP

  END
```

Has Push-Button Game Design Arrived?

Selecting shadow colors for a specific background in MADS. The character's shadows are different in each scene—to match the background art.

If you think about it, what are the various language compilers used by the software industry besides virtualized machine tools? High-level language compilers, advanced graphic rendering systems, audio routine libraries—these are all tools that help game developers build the parts that compose a game product.

Historically, the complexity and specialization of machine tools has grown as an industry grows. Look at any mature manufacturing industry, and the diversity of specialized tools supporting that industry is likely to boggle your mind. The younger the industry, the less rich are the toolsets you can use to build new products.

In the computer game industry, the pinnacle of the software development "machine tool" is the proprietary development platform. For example, Sierra Online's long-term success is due in no small part to the fact that it ventured into adventure games—a game genre with proven appeal. More importantly, it then developed SCI, its adventure game development system, which it used to mass produce new adventure games.

Other early and somewhat legendary development systems include LucasArts's SCUMM and Microprose's MADS. Recently, there's been more talk about development platforms as everyone looks for a leg up on the competition. Sierra Online recently finished upgrading its aging development engine and redubbed it SCI32, while newcomer Rocket Science drew envy with its ballyhooed Game Science system.

The MADS System

Microprose sold the aging MADS system to Sanctuary Woods, which turned around and overhauled the entire system. "The new system is called M4DS," says Matt Grueson, executive producer and director of the advanced development group. Grueson codesigned MADS when he was still with Microprose. When he and a small development group moved en masse to Sanctuary Woods, they convinced their new employer to purchase MADS from Microprose.

The purpose behind MADS, M4DS, and other development systems is to dramatically increase productivity. By reducing the number of programmers needed to complete any given project, the overall budget for a project is drastically reduced. Grueson is clearly sold on the concept, "I had two goals. I wanted to give the end-user a better-looking, better-feeling game, and I wanted it to be more cost effective for us to produce these games."

"With a development engine, pro-

gramming is where we save the most money. We don't need armies of programmers to make everything happen." Microprose's lushly animated DragonSphere required only one person-year of programming time using MADS, in comparison to what Grueson estimates would take other companies, using less evolved development tools, three to four programming person-years to develop. "Based on what I know about the other development systems, their cost would be 50% to 100% higher than ours. If someone was trying to do everything DragonSphere does, and programming it from scratch, I can't even fathom how much time and money that would take."

Collecting the Parts

To design an effective development system you have to start thinking about any game as a collection of parts. A game consists of programming and content. Spend any time talking to designers of these development systems and the most common phrase you will hear is "asset management." Assets are your content, your media: PCX files, bitmaps, WAV files, and the like.

On the software side, a game contains a collection of routines or functions that perform specialized tasks on different media types. The road toward creating a solid development system begins with a good library of such routines, both for graphics and sound.

Get enough solid routines together, encapsulate them within an interface or within a highly abstracted (easy to learn and read) scripting language, and you have the core of a development system. The benefits are numerous. This structure forces you to reuse software assets, which saves money, and the routines that go into the development engine as parts tend to receive extra attention and become highly optimized over time.

Ideally, programmers can then focus on improving and adding new parts to the development engine, while (less expensive) teams of game designers and artists develop titles. In reality, programmers are still an integral part of any development-system-assisted team, even if they're there only to make the game do new tricks.

The Ideal System

The ideal development system abstracts as much of the programming as possible so that artists and designers can use high-level, low-learning curve tools to build the game. "The important thing we kept in mind as we developed these tools was the programmers working on these tools were not the users. The customers were the production team. We wanted each part of the team to be able to work with the system in the way that was natural to them. The artists could do the art with the system, the game designers could do the writing and game design, and you didn't have programmers doing everything," says Grueson.

M4DS is a great example of how a fully developed, well-planned development works. The M4 connotes four Ms: multimedia, multiplatform, multiplayer, and multipoint. Building on MADS, Grueson's team adds multimedia extensions, primarily streaming data handlers that let the system handle lossless video data. Says Grueson, "The data streamer allows us to integrate audio, video, and collision data, like Rebel Assault did—

David Gerding

Fledgling though it may be, the game industry is more than ready for Sanctuary Woods's M4DS and Rocket Science's Game Science, two systems that herald the onset of push-button game design.

ROLLING YOUR OWN DEVELOPMENT SYSTEM

Microsoft has released three Windows extensions that, taken together, could easily form the core of a simple development engine and there for the taking. Sure, everyone knows about WinG, the quasi-double-buffering routine that speeds up graphics blitting under Windows. But go online and get ahold of WinToon, a set of routines that lets you superimpose a sprite on a window playing an AVI file. Also, there's WaveMix, which will let you mix several WAV file sources in real time for multilayered sound effects. Of course, you must already be a Windows programmer or be willing to learn. Microsoft is providing these extensions to get developers to develop games for Windows, and they really are worth a look.

Also, be sure to check out Klik and Play by Francois Lionet and Yves Lamourex. It's a consumer-targeted "game creator" that will teach you a lot about how to think of action games in terms of components. It's way cool.

Finally, Director 4.0 has been released. If you're a PC or Windows fanatic, is new Windows version is a powerful, cross-platform development system in its own right, though you will likely need to learn Lingo, its scripting language, to do anything approaching a commercial-quality game.

almost any type of data that we want to put into the stream. It's fully virtualized. It even handles laying out the files in the correct and optimal order for the best access times."

By keeping many of the requisite game structures represented in data rather than code, M4DS also promises to be highly portable. "We've got a Macintosh engine planned, and we've got most of the work done for a 3DO engine, though we're not sure we're going to proceed with that yet," Grueson says.

Sanctuary Woods has also incorporated a communications sublayer into the engine that will allow for distributed games where multiple players can interact. "The system is designed so that the world within it can be completely virtual. You can have players or nonplayer characters... all interacting within the same game universe. These 'points' can be hooked up via modem, network, whatever. Because the communication layer is completely separate, we can write a communications driver for practically any kind of communication type we might need, including interactive television," says Grueson.

The Building Blocks

The main components of M4DS include a Sprite Editor, Animation Editor, Scene Editor, Object Editor, and Conversation Editor. The Sprite Editor lets artists turns a series of sprites into any one of 24 proprietary formats that trade off various levels of compression depending on the functionality of the sprite required within the game.

The Animation Editor lets artists take backgrounds, sprites, and sounds and define the interaction between these assets. Explains Grueson, "If we have a sprite series of a moving car, a bouncing ball, and a kid walking down the sidewalk, plus a background of a street, the animation will pull all those components together and optimize each of the sprite series to get the most animation in the least amount of disk space."

Every asset is given a specific name, and those names are available to all programmers, designers, and artists. When designing a development system, it's a good idea to follow Grueson's lead and require common file naming conventions so that every member of the development team can quickly assess the contents of an asset by looking at the name.

The Scene Editing tool lets the designer and artists lay out all the elements that compose a particular scene. Because M4DS is primarily a graphic adventure development system, the metaphors and organizing principals it employs follow the structures and aesthetic typical to the adventure game genre. For example, the active screen elements or hot-spots common to all traditional graphic adventures get plenty of attention in M4DS, including a hot-spot properties editor.

The organizing components of any graphic adventure are collections of "scenes," and, subsequently, Scene Edit is where most of the graphical work gets done, reveals Grueson. Even though a development system is usually built upon an organizing metaphor such as scenes, it should also help artists and designers sweat the details, such as allowing artists to match a characters drop-shadow to each specific background.

As for the core game logic, designers working on M4DS use a proprietary scripting language modeled closely on C. According to Grueson, "We chose that because you don't really have to train everybody in it.... Our programmers know it, and the syntax is already pretty well defined."

Game Science

On the cutting edge of development systems, and offering an entirely different perspective is the Game Science development system from Rocket Science. Rocket Science is doing graphically intensive, high-motion action arcade games, and the organizing metaphor is, appropriately, a world away from Sanctuary Woods's. Most of the work is done in a layout program called Composer that is reminiscent of Macromedia Director. "The Composer lets you lay out the game in the form of a conditionally branching time line. We call it TimeSpace," explains Sean Callahan, software engineer at Rocket Science. Callahan—one of the principal designers of QuickTime while he was at Apple—created the Composer prototype.

"Our main idea was to create a system so that game designers could create and develop without being programmers," says Callahan. "In Composer, the designer can do the layout, do the connections, and display it a reasonable level to see how the branches look and that kind of thing. It even tries to approximate the resolution of what the graphics will look like when its run on the target."

The target is important to Callahan because Rocket Science's ambitious goal is

to make Game Science entirely platform independent and able to compile to a target platform at the push of a button. As such, the system currently handles a limited number of media types, including QuickTime movies, sprites, and digitized audio, though the system's abilities are changing to meet new demands.

"Right now we are just looking at how we are going to let Composer handle three-dimensional graphics. While our current system doesn't support true three-dimensional rendering right now, when we render the three-dimensional images on the SGI, part of the output we include in the file is three-dimensional reference information that is stored in one of the tracks in QuickTime. So, for example, if you're driving down a tunnel and you take a right turn... the car headlights seem to reflect on the wall in the right way because we've got enough of the three-dimensional information about the tunnel in the QuickTime track," reveals Callahan.

Rocket Science is just putting the finishing touches on the PC version of Loadstar, originally compiled for the Sega CD target. But all that hard work will pay off in the future, when subsequent titles are readily "compiled" for the PC in a fraction of the time most companies would spend to port a product. "The trickiest part in getting the compiler to port to the different platforms is figuring out all the timing issues with the different drives and the graphic hardware," says Callahan.

The Argument Against Development Systems

Sure, there are cost advantages to implementing a development system, and the allure of such seemingly elegant production is highly attractive. Is there a downside? Yes and no. The potential downside is that games designed with a common development system will look and play alike. While the cookie-cutter similarity between Kings Quest 4-7 hasn't hurt the games' critical acclaim, sales, or profits, there will always be game purists demanding something new and different. Sierra can rightly claim that each new graphic adventure offers something the previous version did not, but the changes have been incremental, rather than revolutionary.

The trick is to let the game designers' needs define the growth of the development system. The tools that programmers periodically add to the development platform should answer designers' and artists' needs. It's simply a matter of keeping the developmental tail from wagging the dog.

Push-button game design has arrived, and as the toolsets evolve, game artists, designers, and programmers will likely need to press fewer and buttons to see their dreams take form. Paradoxically, as the tools proliferate, it may take greater and greater genius in the future to build a game no one has played before. What a marvelous challenge! ■

David Gerding is a freelance writer who teaches interactive media at Columbia College of Chicago.

Programming in C and C++: The Literature

In the days before computers became a lifestyle and surfing was only done at the beach, you found computer books in dark corners of university libraries under "engineering." Not anymore. Eventually, every bookstore, whether on campus or in a mall, had rows of computer books covering everything from Ada to Zmodem. But where were the hard-core references for programming games?

In the past two years, a surge of books that proclaims to reveal all these secrets has emerged. While none of them will make you an instant game programming superstar, any one of them will prove enlightening in some way. Which ones are worth the time and money? I analyzed seven books on DOS game development according to their various strengths and weaknesses.

Flights of Fantasy

First of the group is *Flights of Fantasy* (The Waite Group, 1993, $34.95) by Christopher Lampton. This 550-plus-page book focuses on developing a 256-color VGA flight simulator using Borland C++. Lampton states up front that the simulator won't compare to a high-end program like Falcon 3.0, but that it will give you the basic rudiments and structure to understand how these programs work.

Most of the book is devoted to the essentials of graphics: designing landscapes, the mathematics of flying, how to remove hidden surfaces, and polygon clipping, for example. Lampton explains clearly about interfaces using joysticks, the keyboard, or a mouse. He even manages to touch upon the process of adding Sound Blaster support to your games.

The only major negative aspect about this book is its limited range of applicability. It may prove too daunting if you try to use the Lampton's book to program something more high-level, such as an arcade game.

For flight simulator fans, though, this is the book. Novice programmers may want to get this for its clarity of writing and its chapter on interfaces. Certainly the information on three-dimensional modeling will be of some benefit for those writing first-person-perspective games.

Verdict: Seemingly limited appeal but well written. Would be of most interest to flight simulator fans.

Creating Turbo C++ Games

Clayton Walnum's *Creating Turbo C++ Games* (Que Corp., 1994, $29.99) is a better book for the novice programmer than *Flights of Fantasy*. The 470-page book covers basic game programming and includes a variety of samples. I really liked that the book uses a low-end C++ compiler (Borland's Turbo C++, which sells for under $80). The games you learn how to develop are not sophisticated—card games, a dungeon quest, versions of Wari and Life—but they serve as a good springboard for more detailed programs, and all use 256-color VGA to look sharp.

Walnum's style is friendly and careful as he explains the various components in constructing a game and using C++. For those new to the language, he even includes a short primer in the back of the book explaining object-oriented programming and classes.

The downside is that the games Walnum focuses on are very simple. Top-

ics like sound, joystick interfaces, and killer animation just aren't in the book, so ambitious game developers may require other sources of information to get to the next level of proficiency. Of all the books I reviewed, this one may be the best if you want to design strategy games. The fundamentals for programming strategy games are here; you just have to figure out how to use them.

Verdict: Great book for C++ beginners.

Tricks of the Game Programming Gurus

The most detailed, information-packed book of the seven books I looked at is *Tricks of the Game Programming Gurus* (SAMS Publishing, 1994, $45.00) by Andre LaMothe, John Ratcliffe, Mark Seminatore, and Denise Tyler. This is a must-have for beginning and intermediate programmers, even if you don't have a CD-ROM player to access the enclosed disc. This 740-page book touches on everything: sound, artificial intelligence, networking, assembly language, graphics, and more. The focus is on developing a 256-color VGA Doom-like game. Since this may be the most difficult type of game to develop, understanding the routines here may make programming other types of games easier.

The writing style tends toward a sophomoric buddy-buddy style at times, but the information, tricks, and explanations more than make up for it. The book presents information well, despite the fact that it starts with the topic of assembly language and then dives into matrix algebra two chapters later. These subjects, which are nightmarish yet vital, are handled well and made very accessible.

The book is not perfect: nowhere is it clearly stated what compiler is recommended, though Microsoft C 7.0 seemed to be the chosen tool (e-mail to one of the authors revealed that Watcom was also used). Many advanced topics, such as converting graphics routines to SVGA, are hinted at but then left as exercises for the reader. No code is included for saving and restoring games, but unlike many of the other books here, at least the authors mention the topic.

The accompanying CD-ROM is loaded not only with source code but many shareware games, such as Doom, and Blake Stone, and useful tools such as DigiPak and MidiPak to help you develop sounds for your programs. Also included is Warlock, a rudimentary Doom-like game whose code you can tinker with.

Verdict: A must-have! I only wish it went into even more detail—like using DOS extenders and memory management. The CD-ROM is loaded with source code, tools, and shareware.

Action Arcade Adventure Set

A newer book is *Action Arcade Adventure Set* (The Coriolis Group, 1994, $39.95) by Diana Gruber of Ted Gruber Software. Her book focuses on developing a 256-color VGA side-scrolling arcade game similar to Commander Keen or Duke Nukem. Ted Gruber Software makes Fastgraph, a set of graphics routines, and the book includes a subset of this package, called Fastgraph Light.

Dean Oisboid

It used to be difficult to find any programming books—much less game-specific ones—in your local bookstore. Now, such texts are the order of the day. Here are the latest books on using C to develop games.

I have two points to make about this book. First, too much space is devoted to explaining the source code of the included tools, such as the sprite editor. Almost one hundred pages of the book are devoted to the tools' code, which is great if you want to modify those tools. However, most of us don't want to know how DeluxePaint or Renderman work—we just want to use them. Walnum also explains his graphics tools in *Turbo C++ Games*, but he takes only 30 pages and then quickly returns to game design.

Second, I feel the book reads like an advertisement for Fastgraph, as all the important functions the book describes come from that product. For example, the other books I reviewed explain how to write a joystick interface, but Gruber's doesn't; she suggests you use the Fastgraph routines for this event. No other help is given on the topic.

By relying on a product like Fastgraph, beginning- to intermediate-level developers don't learn how to write vital functions such as keyboard interrupt handlers. What may be worse is losing the ability—especially for the experienced programmer—to tweak any of the Fastgraph functions if there are speed bottlenecks. The entire book relies so much on Fastgraph that to extract the information of writing a side-scrolling arcade game without it would be darn near impossible.

There is a flip side to this opinion, however. Some programmers prefer this reliance, as packages such as Fastgraph and Fastgraph Light let you can concentrate on game design. The products take care of the difficult graphics routines, document them, and if you encounter problems, you can call technical support. Fastgraph runs on most C/C++ compilers.

Reactions aside, the most interesting readings were the headers for the game programs. The headers explain the structure of the game and its components—and right there you get a ton of information. You can see what kind of thought and programming goes into a side-scrolling arcade game and how the planning evolves. Also of interest are the final chapters on marketing and resources. How to sell your finished product is a topic not really touched by many of the books and the resources listing is a very nice addition. The listing includes sources for custom music, sound effects, magazines that may interest developers, and other contacts.

Verdict: Great or to be avoided, depending on whether you want to use Fastgraph as a primary graphics package.

Programming Computer Games in C

Robert B. Marmelstein's book, *Programming Computer Games in C* (M&T Books, 1994, $34.95) is similar to Walnum's *Turbo C++ Games* in that it describes the development of simple games in depth. The games in this book tend toward the "action arcade" side, putting it somewhat in competition with Gruber's book. However, unlike Gruber's book, development doesn't rely on high-powered graphics routines—and it shows. In fact, many samples use 16-color EGA, which places this book about four years out of date.

While each aspect of game construction gets some mention, I felt the programs and logic were too simplistic and that more efficient coding schemes could be used. Marmelstein's use of bitmapping seems very inefficient when compared to LaMothe or Gruber. In fact, Marmelstein explains various design options and then chooses the easiest option, whereas the other authors might have chosen a more efficient yet difficult option. This book would have stood out two years ago, but in order to compete against the LaMothe tome it needs to include much more detail and ascribe to a higher level of game.

Verdict: Well written but not challenging enough. This could have been a much more useful book if Marmelstein tackled, at least, a VGA platform and more advanced detailed games. When compared to the other books, it is painfully obvious that this is the weakest of the bunch.

PC Game Programming Explorer

In contrast to Marmelstein's and Gruber's books, Dave Roberts' *PC Game Programming Explorer* (The Coriolis Group, 1994, $34.95) takes the coding challenge and spells things out. This 500-page book strives to develop one type of game: a top-to-bottom 256-color VGA scroller, similar to Apogee's Raptor (but more rudimentary in play). Roberts uses Borland C 3.1 (4.0 will also work) as the compiler.

The book is very readable and covers the expected topics—interfacing, implementing VGA graphics and special effects, and sound, which makes it an excellent choice for beginners. But, as with Lampton's *Flights of Fantasy*, the next technical step is up to you; to generalize beyond the type of game demonstrated would require some work.

Ironically, The Coriolis Group published both Roberts's and Gruber's book, and my reaction while reading *PC Game Programming Explorer* was that this is what Gruber should have written. Everything, including a joystick routine, is clearly explained—with no reliance on prepackaged graphics routines.

I greatly appreciated the chapter on what shareware and freeware programming tools to get, and where to get them.

Verdict: Great book for beginners to low-intermediates. The narrow range of topics may prove too limited for more experienced programmers.

Teach Yourself Game Programming in 21 Days

As if one huge book weren't enough for him, Andre LaMothe wrote another one. *Teach Yourself Game Programming in 21 Days* (SAMS Publishing, 1994, $39.99) is another required monster, coming in at 950 pages. Topics are broken into daily chunks and each "day" ends with a summary, questions and answers, a quiz, and exercise. He also throws in weekly reviews: a repetition of information that helps you retain the important stuff. Yes, the answers are in the back of the book. As for the compiler, LaMothe recommends Microsoft C/C++ 7.0 and MASM 5.0.

Unlike his previous book, LaMothe tackles a variety of game types instead of just a Doom clone. In fact, he covers some unique topics:

- *Text adventures*. LaMothe includes a

chapter on designing an old-fashioned text adventure. These games sound easy to program mainly because they've been around for so long. However, the implementation of these games is actually difficult if you want to advance beyond the "Get fish," "Eat fish" stage. He explains the difference between lexical, syntactical, and semantic analyses—that is, parsing input into components, "understanding" the components of the input, and finally acting from the components if it's logical to do so—as he develops the adventure game Shadow Lands. As an additional demonstration and a wonderful bonus, the enclosed CD-ROM contains the source code for Zork 1.

- *DOS extenders.* My gripe about his first book has been partially answered. LaMothe acknowledges DOS extenders and gives recommendations and tips for using them, but still doesn't actually show how to use them.
- *Marketing*. While some of the other books I reviewed also mention marketing, LaMothe explains how to protect your work by copyrighting and registering your program with the copyright office. This is a nice touch.

Since LaMothe has coauthored one of the other reviewed books, I'm inclined to compare the two and see what is missing from this newer book. The most obvious difference is that this book doesn't cover three-dimensional games. In fact, you might want to consider both books as a set—his first book covers three-dimensional games and this one explains everything else. Taken as such, the differences are minimal.

Verdict: Another must-have. LaMothe has again written a book that includes so much information that other books can't match up. The CD-ROM not only has source code, it also has more tools and newer shareware games than the *Tricks of the Game Programming Gurus* disk.

Gardens of Imagination

Clayton Walnum's *Gardens of Imagination: Programming 3D Maze Games in C/C++* (The Waite Group Press, 1994, $34.95) follows up on his *Flights of Fantasy* by exploring the construction of maze games like Doom using Borland C/C++. Unfortunately, Walnum covers much the same ground as LaMothe and his coauthors in their *Gurus* epic—but without as much detail. In fact, the bulk of the 580-page book explores ways to produce a first-person, three-dimensional graphics background. In the final chapter, Walnum rushes through the other aspects that go into a game. Implementing sound isn't even mentioned (although Walnum provides neat code for an automap feature). The book therefore reads more like a How-to graphics manual rather than one for a game.

On its own, the book is readable and useful, covering a variety of advanced topics such as ray tracing, ray and height casting, and the always vital code optimization. Of interest is the inclusion of Persistence of Vision, an incredible graphics generator and programming language, that has quite a following of its own. Walnum uses it to produce various parts of the maze background but even this interesting program is not enough to save the book. The focus stays too long on producing different maze types and snazzy background generation and not enough on game development. The book should have been called *Producing 3D Mazes in C/C++*.

Verdict: Bad timing kills this book. A year ago it would have been a groundbreaker. Though it covers similar material, it can't compare to *Tricks of the Game Programming Gurus*, especially when you factor in price and the quantity and quality of programs on the enclosed disk or CD-ROM. Beginners will not suffer, however, if this is the only book available at their bookstore.

A Final Word

All these books provide some good information. If you can afford just one book, invest in either of the LaMothe books; both give you a lot of information, code, and programs for your money and are the most current in terms of techniques. (*Teach Yourself Game Programming in 21 Days* does have the newer versions of many shareware games, though.)

For the complete beginner awed by the prospect of wading through either of LaMothe's books, my recommendation is Roberts' *PC Game Programming Explorer* for the depth and clarity of writing. For C++ aficionados, Walnum's *Creating Turbo C++ Games* gets the nod. It covers a variety of game types in a well-written manner.

One caveat about these books is that they all claim to reveal the deepest, darkest secrets of game programming, which each one does to an extent. They all reach a certain point before waffling. None of the books dared to show how to program SVGA or implement DOS extenders (and try to write a decent Doom clone without using some sort of extender). Many of these advanced areas were left as exercises for the reader to figure out. Also left as an exercise, surprisingly, is how to implement save and restore routines. Some of the books mentioned the topic but none offered any real code.

Realistically, as games continue to push the envelope of hardware, a prospective game programmer just starting out will quickly hit a wall. Indeed, the in crowd of game developers will still remain "in" when it comes to the high level, SVGA, network and or modem-capable, AI-intensive, next generation games. The fact that there is so much to learn and that new techniques are continually invented can be daunting to the beginner. But, with work, patience, and perhaps a little luck, the darkest secrets of game programming will be revealed. ■

Dean Oisboid, owner of Garlic Software, develops database applications and is a game designer. He can be reached via e-mail at 73717.2343@compuserve.com. or through Game Developer *magazine.*

Tie Fighter, Part II

Wayne Sikes

After an initial, general look at LucasArts's Tie Fighter game engine, Wayne Sikes digs deeper. This month, he scrutinizes the game's data files, as well as the structure of the Defender of the Empire add-on mission set.

This month, we conclude our review of TIE Fighter by LucasArts Entertainment Company. We've already looked at the TIE game engine in general, and I wrapped up last month by summarizing some of the data I found in the pilot file. This month, we'll go through some of the data found in the TIE mission files. LucasArts released Defender of the Empire, TIE Fighter's add-on mission disk set, just as I began writing this review, so I'll also cover some of the mission data I found in these new missions.

The data files in the Defender of the Empire mission set were somewhat different from what I expected. The add-on missions for X-Wing, the first game in the Star Wars series, contained new game engines (the primary executable plus two overlay executables) in each mission set. TIE Fighter's mission set consists primarily of the new mission files plus some graphics and sound files for the new Missile Boat vehicle. The absence of a new game engine in the TIE add-on missions implies that the original game engine had the data for add-on missions built into it—the add-on mission sets will activate this data as needed. LucasArts's strategy of creating only one game engine is good in that it forced the company to plan out the entire TIE game series from the start. This makes for cleaner and more reliable executables—as compared to game engines that must be updated with each release of new missions. Also, by not distributing new game engines with each add-on mission set, LucasArts saves money on duplicating and distributing mission disks, because the game engine files are quite sizable.

Tie Fighter add-on missions don't modify the original game engine, they just activate data already incorporated in the engine.

Listing 1. General Flight Group Parameters

```
STRUCTURE          DATA      DESCRIPTION
OFFSET (DECIMAL)   TYPE*

0-11               byte      Flight Group Name.  12-byte null-terminated array.
24-35              byte      Cargo 1 text.  12-byte null-terminated array.
36-47              byte      Cargo 2 text.  12-byte null-terminated array.
48                 byte      Special Craft Position.
50                 byte      Vehicle.  1=X-Wing, 2=Y-Wing, 3=A-Wing, etc.
51                 byte      Vehicles Per Wave.
52                 byte      Starting Configuration. 0=Normal -> 8=Shields off
53                 byte      Weapons. 0=None -> Magnetic Pulse
54                 byte      Beam Weapons. 0=None -> 3=Decoy Beam
55                 byte      Affiliation. 0=Rebel, 1=Imperial, 2=Neutral, etc.
56                 byte      Artificial Intelligence. 0=Novice -> 5=Super Ace
58                 byte      Talk Flag. 0=Talk off. 1=Talk on.
60                 byte      Formation. 0=Vic -> 9=Vertical
64                 byte      Number Of Waves.
66                 byte      Player Position.
73                 byte      Difficulty. 0=All Levels -> 5=<Hard Levels
96                 byte      Arrival Mother Ship Flight Group.
97                 byte      Arrival Method. 0=Hyperspace. 1=Mothership
98                 byte      Departure Mother Ship Flight Group.
99                 byte      Departure Method. 0=Hyperspace. 1=Mothership

*  "byte" references an unsigned character.
```

TIE Mission File Overview

The TIE Fighter mission files have a .TIE suffix. Refer to "TIE Fighter, Part I" (Chopping Block, Feb./Mar. 1995) for information on how the battle and historical mission files are named as well as for listings of the mission files found in the original game distribution. The TIE mission files range in size from about 2,000 to 22,000 bytes. Each mission file contains essentially all the flight group data, briefing text, radio messages, mission objective summaries, mission accomplishments, enemy opposition summaries, and instruction or warning messages that appear during a mission. X-Wing, on the other hand, had separate mission and briefing data files, and its mission and briefing text was nowhere as detailed as TIE Fighter's.

The data for each flight group is contained in a 292-byte structure. All flight group structures are grouped together with the first structure beginning at offset `1CA` (hex) in the mission file. The flight group structures, and hence the flight groups, are ordered in a one-up manner with the first flight group structure in the file as "flight group 0".

The TIE mission data is one of the most complex mission structures I have seen. Due to this complexity, I will be able to discuss only a few of the mission details. Hopefully, the mission parameters I cover here will give you a broad idea of the types of data used by the TIE game engine for setting up missions.

General Mission Parameters

Listing 1 gives several parameters contained in the flight group structures. As you can see, each structure contains a large amount of flight group data, and we are just getting started! The flight group Name, Cargo 1, and Cargo 2 data are standard C `null`-terminated string arrays. The last character of each array must be `null` (0), so each string can contain up to 11 characters.

The Vehicle byte specifies the vehicle used by the flight group. Each game vehicle is referenced by a specific value—a value of 1 specifies an X-Wing, 2 is a Y-Wing, 3 is an A-Wing, and so on. TIE Fighter has many more vehicles than X-Wing. The Vehicles Per Wave variable specifies the number of vehicles that will appear with each wave of craft, and the Number Of Waves byte specifies the total number of waves allowed to appear.

Several parameters specify the condition of the flight group vehicles when the group first appears in the game. The Starting Configuration specifies normal, extra weapon, damaged, and special (shields off, hyperdrive off) configurations. The Weapons variable gives the default weapon load and the Beam Weapons details any beam weapon loadout. The Beam Weapons data is especially fun to alter because in the original game only Darth Vader had a Decoy Beam weapon. Now you can give yourself one too!

The Artificial Intelligence byte tells how good the pilots in the flight group are. You can set this level from Novice (not very good) to Super Ace. (I usually avoid the Super Ace settings because I get destroyed very quickly.) The Affiliation variable sets the allegiance of the flight group (such as Rebel, Imperial, or Neutral), and the Talk Flag toggles your ability to talk to or command the flight groups. (I usually turn this flag on when programming enemy flight groups so I can tell them to "go home" when I'm being beaten badly.)

Every flight group can arrive and depart via hyperspace or a mothership. The Arrival and Departure Mother Ship Flight Group variables specify the flight group designated as the mother ships. The master controls for how vehicles arrive and depart are in the Arrival and Departure Method bytes. If you program a flight group to jump into hyperspace in the game, the vehicle you select for the flight group *must* have hyperdrive engine capability.

Flight Group Start Conditions

Listing 2 summarizes the flight group structure data that specifies when a group enters the game. I have given two start conditions the arbitrary label of Primary and Secondary Start Conditions. Both start conditions function in the same manner.

Several possible Primary and Secondary Start Conditions exist. These conditions include an "always start" con-

dition (0), "the designated object must have arrived" (1), "the object must have been destroyed" (2), and so on. The start condition depends on actions happening to another object. This object may be another flight group, a certain type of vehicle (X-Wing, Y-Wing, and the like), or a vehicle that has a specified allegiance (Rebel, Imperial, and so on). The condition may also depend on a range of flight groups (flight groups 1 to 6, for example). The Primary and Secondary Dependency Type variables specify the type of object used for the start condition and the Primary and Secondary Start Data variables contain the data for the object. If the object is specified as a flight group, the data variables will contain the number of the flight group. If the object is a type of craft, the data will specify the craft in the dependency condition.

In addition to all the Primary and Secondary conditions, there can also be time delays that prevent a flight group from entering the game for a specified time interval. The Start Minute Delay counts in units of minutes, and the Start Second Delay counts in intervals of five seconds per data increment.

The Primary And Secondary Logic Switch is interesting in that it specifies whether the player or game must accomplish *both* the Primary and Secondary Start Conditions or just either *one* of them. When you set this switch to 0, a logical AND condition is specified and the game or player must meet both start conditions. A value of 1 specifies a logical OR condition and only one of the start conditions must be met for the flight group to enter the game.

Commanding The Flight Groups

Commanding the flight groups is one of the most interesting yet complex areas of the TIE game engine. A lot of forethought went into the design of the algorithms used for the orders that can be given to a flight group.

Each flight group can have primary, secondary, and tertiary orders. The flight group structure stores each order as an 18-byte structure with the three orders structures occurring sequentially in the flight group structure, beginning at offset 104 (decimal) and ending at offset 157 (decimal). Listing 3 summarizes some of the parameters found in each of the orders structures. Notice the detail given to each order.

The Order variable contains at least 35 commanded orders. A value of 0 commands the flight group to remain stationary, 1 instructs it to fly away, 2 commands it to fly a loop, and so on. Any details associated with the order, such as time constraints or the number of loops to fly, are found in the Indicator1, Indicator2, and Indicator3 bytes. The Commanded Speed byte details the velocity for the flight group while it is carrying out the order. The speed values increment in units of 10% of vehicle velocity.

Up to four targets can be associated with each order. Each of these targets is specified in the Target X Data and Target X Type variables (where X is targets 1 to 4). The Target X Type variable specifies the type of target and the Target X Data variable contains the data for the target. Among the available target types are flight groups, types of craft (X-Wing, A-Wing, and so on), or craft having a specified allegiance (Rebel, Imperial, and the like). The Global Player is a special target type. The Global Player type and its associated data are "modifiers" of other target objects (I'll discuss global data types in more detail later.). The data for each target (Target X Data)

LucasArts's single game engine-forced the company to plan out the whole game from the start, which makes for more reliable executables.

Listing 2. Start Condition Data

STRUCTURE OFFSET (DECIMAL)	DATA TYPE*	DESCRIPTION
74	byte	Primary Start Condition. 0=Always. 1=Arrived, etc.
75	byte	Primary Dependency Type. 0=no dependence, 1=flight group dependence, 2=vehicle type dependence, etc.
76	byte	Primary Start Data.
78	byte	Secondary Start Condition. 0=Always. 1=Arrived, etc.
79	byte	Secondary Dependency Type. 0=no dependence, 1=flight group dependence, 2=vehicle type dependence, etc.
80	byte	Secondary Start Data.
82	byte	Primary And Secondary Logic Switch. 0=AND. 1=OR.
84	byte	Start Minute Delay.
85	byte	Start Second Delay.

* "byte" references an unsigned character.

Listing 3. The Orders Structure

STRUCTURE OFFSET (DECIMAL)	DATA TYPE*	DESCRIPTION
0	byte	Order. 0=remain stationary, 1=fly home, 2=fly loop, 3=fly loop and evade, 4=rendezvous, etc.
1	byte	Commanded Speed. 0=stopped -> 10=100% velocity
2	byte	Indicator1. General timer, loop counter, etc.
3	byte	Indicator2. General timer, loop counter, etc.
4	byte	Indicator3. General timer, loop counter, etc.
6	byte	Target 1 Type. 1=flight group, 2=type of craft, 5=allegiance, 7=global, 8=flight group range
7	byte	Target 2 Type. 1=flight group, 2=type of craft, 5=allegiance, 7=global, 8=flight group range
8	byte	Target 1 Data. flight group, vehicle, etc.
9	byte	Target 2 Data. flight group, vehicle, etc.
10	byte	Targets 1 And 2 Logic Flag. 0=AND, 1=OR
12	byte	Target 3 Type. 1=flight group, 2=type of craft, 5=allegiance, 7=global, 8=flight group range
13	byte	Target 3 Data. flight group, vehicle, etc.
14	byte	Target 4 Type. 1=flight group, 2=type of craft, 5=allegiance, 7=global, 8=flight group range
15	byte	Target 4 Data. flight group, vehicle, etc.
16	byte	Targets 3 And 4 Logic Flag. 0=AND, 1=OR

* "byte" references an unsigned character.

can be a flight group number (assuming Target X Type specified flight groups), a vehicle type (assuming Target X Type called for a type of craft), an allegiance (assuming Target X Type called for allegiance types), and so on.

If this discussion of targets hasn't already been complex enough, let's add in the logic switches and global types. In Listing 3, the data for targets 1 and 2 is grouped together, as is the data for targets 3 and 4. These data were not placed in the orders structure randomly. Targets 1 and 2 can be grouped together into a single target specification, and the same goes for targets 3 and 4. The grouping is done by using two Boolean flags—the Targets 1 And 2 Logic Flag and the Targets 3 And 4 Logic Flag variables. A value of 0 commands that the two targets function in an AND manner (both groups work together), and a value of 1 signals an OR condition (the two target groups work independently). As I previously mentioned, the Global Player target type is a modifier of other targets. It is used in conjunction with these logic flags to modify the actions a flight group takes against a target.

Listing 4. Win and Bonus Variables

STRUCTURE OFFSET (DECIMAL)	DATA TYPE*	DESCRIPTION
158	byte	Win 1 Condition. 1=created, 2=destroyed, etc.
159	byte	Win 1 Detail. 1=50%, 4=special vehicle, etc.
160	byte	Win 2 Condition. 1=created, 2=destroyed, etc.
161	byte	Win 2 Detail. 1=50%, 4=special vehicle, etc.
162	byte	Possible Win 3 Condition.
163	byte	Possible Win 3 Detail.
164	byte	Bonus Condition. 1=created, 2=destroyed, etc.
165	byte	Bonus Detail. 1=50%, 4=special vehicle, etc.
166	signed char	Bonus Points. 1 increment = 50 points.

* "byte" references an unsigned character.

Let's look at an example to help simplify this targeting information. Let's assume you are flying a TIE Fighter and you want to program a flight group to attack all TIE Fighter vehicles—that is, all TIE Fighter craft except yours. How would you do this? First of all, program the flight group with the Attack Flight Group orders (give the Order byte a value of 7). Let's arbitarily use targets 1 and 2 for this order. We would set Target 1 Type to specify a type of craft (a value of 2 commands for a type of craft), and next we would set Target 1 Data to a value of 4, which would specify TIE Fighters.

If we stopped right here, then all TIE Fighter craft, including yours, would be attacked. Set Target 2 Type to 7, which invokes the Global Player type, and set Target 2 Data to 10 (decimal) which is the Global Not Player option. Finally, set the Targets 1 And 2 Logic Flag to 0 to force the AND combination of targets 1 and 2. Now all TIE Fighter craft except yours will be attacked. I realize that I have left out lots of detail here, but I hope you understand some of the logic used in commanding flight groups. (I said previously that the orders system in TIE Fighter is complex. Was I right?)

Winning the Game

There are Primary (Win 1) and Secondary (Win 2) win conditions that players must meet to win a mission. Refer to Listing 4 for a summary of the win and bonus conditions specified in the flight group structure. You'll see that I've included a possible Win 3 condition, but I do not have much evidence that indicates this condition exists.

The Win 1 Condition and Win 2 Condition variables contain the conditions that must be met. The values in these conditions are the same as those found in the start condition variables. A value of 1 specifies that the group must have arrived, 2 means the group must be destroyed, and so on. The Win 1 Detail

and Win 2 Detail bytes function as modifiers of the conditions. For example, if the condition specifies that a flight group be destroyed, the detail might modify this situation to mean that only the special craft in the group be destroyed to win.

The Bonus Condition byte functions the same as the win condition variables and the Bonus Detail functions the same as the win detail parameters. If the bonus condition and detail are met, bonus points can be awarded. The Bonus Points variable is a `signed char` value. Each increment in the value represents 50 points. Since the bonus variable is a signed value, note that negative or penalty bonus points can occur. For example, if the bonus condition on your mother ship is that it be destroyed, a penalty of -10,000 points may be "awarded" if the enemy gets past you and takes out your mother ship.

Want to Edit Some Missions?

When analyzing the more detailed aspects of a game, I frequently write utilities that help with my analysis. The end result of my analysis of the TIE mission structures is a routine called TIEDIT. If you've read this far, then you know how complex the TIE mission data is. For this reason, I opted to write TIEDIT in MS Windows because it is relative easy to implement list boxes, edit boxes, and the like in Windows. The sheer number of mission items I could edit was enough to make me abandon any thoughts of a DOS-based TIE mission editor. Completion of TIEDIT took longer than I anticipated because every time I tested it I would get "hooked" by the new missions I was creating, and then I'd spend too much time playing the game! TIEDIT.ZIP is on CompuServe in the Flight Simulation Forum (GO FSFORUM), Space Combat Library.

We Have A Winner Here!

As I've said before, LucasArts gaming products are generally well written and executed, and TIE Fighter is no exception. The level of detail LucasArts developers have given to TIE Fighter's mission structures is fantastic, to say the least. This dedication to a product will considerably prolong TIE Fighter's life on the shelves via more add-on mission disks. I'd really like to see LucasArts release some form of the mission editor used in creating the retail missions. Obviously, the company might want to wait until it's done creating more add-on missions, but a retail LucasArts TIE mission editor could possibly extend the life of the game by years. ■

Wayne Sikes has been a computer hardware and software engineer for the last 10 years. He has an extensive backgrond in C, C++, and assembly language programming. He also has several years experience as a computer systems intelligence analyst, where he specialized in deciphering and disassembling computer code on classified government projects. He has written numerous computer gaming help utitlities. You can reach him via e-mail at 70733.1562@compuserve.com or through Game Developer.

It's a Sim, Sim, Sim, Sim World

Alexander Antoniades

When it was first released, SimCity, from Maxis, was the sleeper of computer games. Now, numerous sequels later, Maxis is embarking on a new venture that will spawn a whole new SimWorld.

As computer game companies expand, they often tend to lose their identity. The game responsible for the company's initial success gives way to other projects, and soon several creative teams are working on different games, some of which are successful and some of which aren't. The next thing you know they're hawking some Doom clone and you say to yourself, "This is the company that did x?" This is not the case with Maxis.

At Maxis, the apple never falls far from the tree. And that tree, of course, is SimCity, the game of civic management that has not only gone on to sell more than one million copies across a number of platforms, but has also inspired countless articles in famous periodicals, imitators aplenty, and everything short of a Nobel prize. SimCity was followed by SimEarth, SimAnt, and SimLife, which extended the Sim product line and cornered a certain mental niche among gameplayers.

Indeed Maxis is one of the names people are likely to invoke when they talk about the positive aspects of computer games. The "software toy" that educates as it entertains became Maxis's hallmark, and the company thrived on it. Using the magic formula that combined provocative games with well-written manuals, excellent distribution, and some of the highest registration rates in the industry, Maxis was often able to guide even the most complex game into six-figure sales.

In 1994, after releasing the much anticipated sequel to SimCity, SimCity 2000, the Orinda, Calif.-based Maxis was becoming one of the largest computer game companies in the industry. It had grown to over 130 employees in five years and was moving toward going public. But first there were two development issues Maxis had to deal with.

First, Maxis was still primarily a Macintosh developer in what was rapidly becoming a PC market. Co-founder Will Wright was a diehard Macintosh fan who developed most of Maxis' games first on his favorite development environment, the Macintosh. Unfortunately the sales breakdown favored the DOS market by a three-to-one factor, so when DOS versions of Maxis's games were often six months to a year behind the Macintosh versions the company ended up losing revenue.

This is a problem that Brian Conrad, technical director of Maxis, has had to deal with. Maxis developed SimCity 2000 for DOS and Macintosh simultaneously, but, due to the complex nature of DOS platforms and trouble with the VESA video drivers, the DOS version still ended up three months behind the Macintosh version.

For the long term, Maxis hopes to come up with a cross-platform development environment that favors what it thinks will be the big market over the next couple of years—Windows. In the meantime its developers still use the standard DOS development environment consisting of Miles Design sound drivers, Tenberry's DOS Extender, and Watcom C++ 10.0. The company is very close to moving to NT with Visual C++, but until Microsoft can guarantee that what works under Windows will work under NT, Maxis will hold off. Conrad expects that the company's last DOS products will be

the upcoming SimTown and SimIsle, unless running three-dimensional graphics under WinG invites a huge performance penalty.

While Maxis addressed the Macintosh vs. DOS issue, it was still struggling with another one—namely, the interlocking nature of Maxis' games, and the future of its cash cow, the SimCity series. While SimCity 2000 was a smash hit, Maxis couldn't see a clear path for improving the series by adding new games to it. But most Maxis games occupied some of the same intellectual real estate, which made them natural candidates to extend into each other. For example, one of the central strategies a player can use in SimCity is to develop a mass transit system to help the city grow. In A-Train, a game Maxis imported from the Japanese company ArtDink, the goal is to help a community expand using a commuter rail line. In the same vein, the monster towers of the future in SimCity 2000 are similar in nature to those players build in another Japanese game imported by Maxis, SimTower.

Thus the SimWorld project was born. The goal of this project was twofold. It would allow Maxis to create future games with the capability to extend and merge into other preexisting and future games. And it would leverage SimCity and the millions of hours worth of playtime that people have invested in SimCity saved games into new markets.

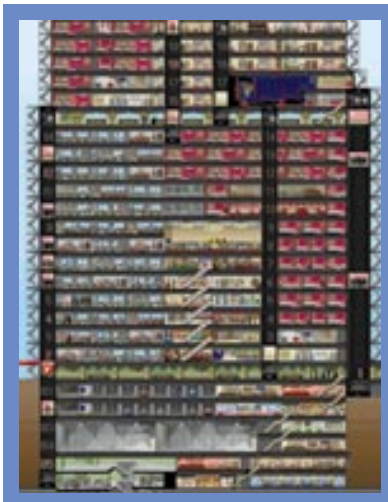
SimWorld

"Most of the project is formalizing the interface," says Jim Mackraz, director of Maxis' newly formed Core Technology Group. Using his experience as a systems designer (Mackraz worked on the Amiga's Intuition interface), he and his group are trying to set the ground rules for the SimWorld project so that developers can integrate them into future Maxis games as they are making them.

The first version of the SimWorld architecture will be based around games similar in scale to SimCity 2000's data structure with one more layer of detail. This first stage of SimWorld will rely on static data exchange—that is, a foundation of users' saved games. Maxis would like the first exchange to occur between Will Wright's "Project X," the likely successor to SimCity 2000, and a three-dimensional helicopter rescue game that uses the same data set to create scenarios. In this case, players of Project X can use the helicopter game to perform rescues in cities they've created in Project X.

For the next stage of the project, Mackraz envisions developers breaking down games into modular programs that could pipe data to one another. For example SimCity 4000 might contain a SimCity data server that would feed an editor client and a three-dimensional viewer, and a supplemental product like SimRail would include more data objects and separate train builder client.

Mackraz stresses SimWorld will not be a low-level infrastructure or an application framework for games, but various games pieces that are connected. He is currently leaning toward the lowest common denominator of object technology, Microsoft's OLE and COM, but hasn't ruled out other technology. Maxis sees this as the framework that will allow its developers to write common tools that can be used by any SimWorld game. Developers may also be able to build translation layers that can exchange SimWorld data with other gaming systems.



SimTower, a Japanese game, is just one of the many titles currently out that falls under the Sim umbrella.

As far as developer relations are concerned, the SimWorld project will go through three stages. SimWorld will first appear as an internal set of development libraries; next, Maxis will release a software development kit to close partners; and finally, the company will put out a software development kit for any interested developers. Being an old operating systems guy, Mackraz is worried about letting a host of developers work on SimWorld because someone might create a killer app that cheats on the specs—and Maxis would have to continue to support it.

Everyone at Maxis agrees that driving this technology should be great games and not the other way around—the industry should never put technology over game play. Maxis's goal is that all of the SimWorld components should feel like one world, and a synergy will exist between the games developers create using those components. "We're going to title our way to an operating system," says Jim Mackraz, indicating that SimWorld will be extended with each new game.

Project X

SimCity was born while Will Wright was working on a Commodore 64 action game called Raid on Bungling Bay for Broderbund in 1985. He became interested in the utility that he used to build the islands that were going to be bombed in the game. After he finished the game, he developed the first version of SimCity for the Commodore 64. Unfortunately, Broderbund passed on the game, so it was never released.

In 1987, Wright bumped into Jeff Braun, who wanted to start a game company and was interested in helping develop SimCity for the new 16-bit computer systems. Wright started working on SimCity for the Amiga and Macintosh. When SimCity was released for the Macintosh in 1989, *Time* magazine did a cover story on it, and the rest is history.

Reflecting on critics and reporters using SimCity as a socio-economic urban divining rod, Will Wright says, "I don't think anybody can build an accurate model of something as complex as a city. It's just too chaotic." But somehow that never stops him from trying.

His latest game, code-named Project X, is perhaps the closest he has come yet. It goes deeper than any of the previous SimCity projects—in this game the player

will be able to interact in the city he or she builds. I got to see this firsthand. Using an existing SimCity 2000 saved game, Will Wright demoed some of the working code for Project X.

The game starts out on the same detail level as SimCity 2000 and then zooms down until one SimCity 2000 tile fills the screen. The player then creates people who will walk around and interact with each other. As the central character walks, the screen scrolls and follows his or her path. This person will eventually meet other "simpeople," randomly determined by the surrounding socioeconomic conditions based on the SimCity 2000 data.

Project X will likely determine the data structure that the first version of SimWorld will use. That data structure is such that managing the level of detail will be one of the biggest challenges in this project. Wright estimates that in order to expand the current SimCity 2000 model to where the player can walk on every floor of every building will take about 30MB to 40MB worth of datasets, which definitely makes this game CD-ROM based. The data will be loaded on the fly as the character walks all the way through game—the only data saved in the actual game will be where specific interactions occur. For example, if you stop to buy a newspaper, and the newspaper agent tells you his name is Joe, that will now be part of the saved game.

The danger seems to be the inherent difficulty in saving large games. But Wright remains confident that he can keep the data sufficiently compressed that size will not be an issue. The current model would make a completely data-rich Project X saved game, that is, one where everything had been tracked or modified, at 3 gigabytes worth of data. But Wright says playing the game every day for a month would only result in a saved game that was about 1MB in size, which isn't bad for that amount of playtime.

Project X will put an important component of the SimWorld concept to the test—Maxis's object-oriented data structure, which the company hopes to use to extend its future games. Static objects in Project X can be embedded with behavioral characteristics for the simpeople. The simpeople here aren't bitmaps, but stored pieces of component-based geometry with basic rules governing how they move. So objects can be introduced into an environment that wasn't designed to support them. For example, a soccer ball could contain the rules of soccer, so when the soccerball object appears, a simperson would simply start playing soccer.

Project X will provide the early foundation of the SimWorld model, and its success or failure will guide Maxis into what is likely the most ambitious project this young industry has seen. It looks like Maxis, using game play and not technology as it guide, will build the first integrated game environment. But in complex data models like the computer game market, no simulation can accurately predict what will really happen. ■

Alexander Antoniades is Game Developer's *editor-at-large.*

Would You Like Virtual Buttery Flavoring With That?

Hand-animated sequences distinguish a routine fighting game and reward players for clearing each level. Pictured here is Mutant Rampage: Bodyslam, by Animation Magic for CD-i.

As the processing power of computers and consoles becomes ever greater and the expectations of game designers and consumers alike rise yet higher, the artist is forced to wade waters of increasing depth, treading further from shore while the comforts of two-frame sprite animation and tiled backgrounds recede in the distance.

This is not a bad thing—far from it. Artists can now exercise greater creativity in game creation, flexing muscles that had previously been underused, given the limited graphics capabilities of yesterday's games. Yet at the same time, this newfound "freedom" brings with it new challenges. Consumers want flash and sizzle. Glossy, cinematic title sequences and transitional animations are now all but obligatory. And with multimedia PCs already in an estimated 10 million U.S. households, the demand for "interactive movie" titles is growing.

With digital entertainment acquiring more and more the characteristics of cinematic entertainment, game artists are being called upon to adopt the roles of the traditional film crew and to reinterpret on their computers the functions of a movie set. Whether these complex "virtual sets" will be used throughout gameplay or only in title and in-between sequences, their creation requires a diverse array of skills ranging from set designer and propmaster to lighting technician and cinematographer, not to mention the technical demands of creating full-fledged animation. You can skirt the significance of these roles, certainly—you can approach an animation without giving much consideration to viewpoint or composition or quick cuts—but the cost is in visual quality, and these days that can be a high price to pay.

Let's Go To The Movies

If we are, then, to adapt the roles of myriad film technicians to our task as computer artists, it is appropriate that for inspiration and guidance we turn to the movies in which their work comes together. We can benefit from their hard-earned experience as we usher in a new era of digital entertainment.

Watching a good film is an immersive experience. The willing viewer is carried along by the plot as by a current. Yet, though the storyline and action may seem to flow naturally, the film itself is an assemblage of artificial devices painstakingly combined for effect. In our quest to incorporate similar narrative sequences in the animations so common now in computer games, we would do well to note how and why effective cinematic storytelling works.

Few moviegoers dissect a film as they watch it, preferring to suspend disbelief and enjoy the show. But cinematic technique can only be understood by *observing* a film rather than merely being swept along by it. Observe the cuts within a scene and note that though the view may switch from close-up to medium shot to tracking shot and back to close-up, the overall effect is still fluid; try to determine the placement of lighting required to achieve *just that effect*; pay attention to camera position and angle and the way they affect the mood or meaning of a shot. When the machinery behind the illusion is revealed, the magic

of cinema's ability to transport the audience becomes all the more miraculous, the artistry of the film crew all the more admirable.

The aesthetic considerations of the digital entertainment artist mirror those of the filmmaker, whether that artist is creating animations in two dimensions or three. For the most part, the creative decisions we must make transcend technique or the actual tools an artist may use to create a scene; aesthetic considerations are the same regardless of the method we use to bring them about. Whether to work in two-dimensional or three-dimensional graphics, then, is chiefly a stylistic preference. Both require a demanding skill set and both rely, ultimately, on the artist's masterful control of the elements that combine to make a scene.

If It Was Good Enough For Walt ...

At Animation Magic, a fairly typical fighting game (CD-i's Mutant Rampage: Body Slam) was jazzed up by means of between-play interviews with the combatants, à la TV wrestling. The artists handled the animation using traditional techniques—pen and paper—then scanned the results. Later, they used the computer to clean up and "paint" the hand-rendered frames. The final look is of a comic book in motion, which suits the game to a tee.

Fractal Design Painter offers another two-dimensional solution, with an "onionskin" function that lets you view several still frames simultaneously, like tracing-paper overlays, to aid in rendering animations. You can also create a new image by tracing an existing one, say a digitized photo or frame of film or video. Added to these helpful features is the ability to emulate natural media such as oil pastel or watercolor, creating effects so convincingly "hands-on" you expect to find finger smudges in the corners.

The resulting hand-drawn look of such two-dimensional approaches can be a refreshing and effective departure from the sometimes sterile slickness or chunky pixelization of computer graphics. If you're not already a fluent practitioner of traditional animation techniques, however, this can be a tough way to start trying to flesh out your game graphics. While it certainly speeds up the inking and painting time of old-fashioned cel animation, two-dimensional software provides no help with creating the illusion of space—perspective, foreshortening, lighting effects, cast shadows, and the like are all up to the artist to figure out—and if you elect to reframe a shot to depict it from a different view for heightened mood, you're talking about redrawing from scratch.

What's New

Such limitations are nothing new; it's essentially the way all hand-rendered art has been made throughout history. However, with the arrival of three-dimensional modeling and animation programs, the computer artist can leave behind the drudgy concerns of the draughtsman and concentrate on the creative challenges of visual narrative. And you don't need to sell your spare organs to scrape up the price of a Silicon Graphics workstation to take advantage of the power of three-dimensional animation, either. Programs like Caligari trueSpace, Visual Software's Visual Reality, 3DStudio from Autodesk, and many others are available for a variety

David Sieks

Game animators can often take their cue from what's happening (and been happening) in Hollywood. The creative processes of movie making and game design are strikingly similar.

of platforms at down-to-earth prices, relatively speaking. Some are significantly closer to terra firma than others, but compared to a $100k SGI set-up these all qualify as terrestrial in price while still delivering out-of-this-world graphics effects.

To a greater or lesser extent these programs all offer variations on the same glorious theme; your monitor becomes a window onto a virtual world of *your* creation. Look at it from any angle. Light it brightly or dimly or fill it with fog. Change the textures of its surfaces. Do what you will, then undo it and try it a different way. The power is dizzying.

The range of options is dizzying, too, and a formidable knowledge base is required to make full use of them. A 250-page manual is required, for example, to illuminate just the "new features" in Release 4 of 3DStudio.

In the Director's Chair

Regardless of whether you already have experience with two-dimensional or three-dimensional animation, or whether you are even now wondering what method you'll use to incorporate cinematic sequences into your next game, giving some thought to film art techniques can probably help you add something special to that next project.

Think About Lighting

While a camera certainly needs light in order to capture images, any cinematographer will tell you that lighting in a film serves more functions than mere practical illumination. Perhaps more than any other visual element, lighting sest the mood for a scene. Tension builds in darkness and shadows. Flickering firelight can instantly make a scene appear cozy, or hellish, or both in turn. Shading can transform a face from angelic to monstrous.

It's important for lighting effects to seem naturalistic, but Hollywood learned long ago that realism is secondary to results. In many shots, each major figure in a scene will be illumined by three or more lights: a *key light* for the defining highlights and shadows; *fill light* , used to soften the effect of the key light; and a *backlight* to separate the figure from the background.

A two-dimensional animator can draw in the shading and highlights that seem appropriate. A three-dimensional animator must create them by positioning virtual lights within the virtual set. Three-dimensional modeling programs will allow you to set a brightness value and color for ambient light and position spotlights to customize the illumination of objects and figures. With 3DStudio, you can even indicate where on an object you wish the highlight to appear and position the light accordingly.

Think About Framing and Focus

Viewpoint is a powerful and yet nearly invisible tool in the cinematographer's kit. Careful placement of the camera can underscore the relationships of figures or objects in a scene by visually grouping or separating them. Close-ups can focus attention on a detail or a fleeting facial expression, while an extreme long shot can establish locale or serve to make the clash of two armies seem puny against the scale of their surroundings. A high angle shot makes the subject appear smaller and might suggest weakness, whereas a low angle shot is often used to impart power and stature to the subject.

Such varying viewpoints go unquestioned by the audience—we don't even wonder why, for example, we are apparently looking down on a scene from a corner near the ceiling—because of the natural way in which viewpoint fits narrative structure and works to contribute to mood and meaning. But while the range of camera positions and angles is essentially limitless, their successful usage is a carefully considered decision.

This is worth keeping in mind, as three-dimensional modeling and animation software will enable you to position the viewpoint anywhere in relation to the scene, and it's easy to get carried away. As with placement of lights, the virtual camera can be positioned visually or by means of Cartesian (X,Y,Z) coordinates to dictate a precise location. Involved camera movements are also facilitated by three-dimensional animation software. With two-dimensional methods, character animation is readily manageable but it can be a real challenge to move the viewpoint around or through a scene.

Another subtly powerful tool provided by three-dimensional programs is the ability to affect *lens optics*. Just as a real camera can be outfitted with varying degrees of lenses to affect focal length, so can your virtual camera. The effect is a distortion of perspective, which affects the perception of depth in a scene. The ability to alter the appearance of a view in such a manner is crucial to the filmmaker's art, and, again, it can be extremely difficult to duplicate with traditional, two-dimensional animation methods.

Think About Editing

The art of editing within a scene is perhaps the most significant aspect of narrative film technique, yet it is also the least noticeable. Editing is so intrinsic to the way we expect a scene to unfold that the details of each cut slide by without registering on the consciousness of the viewer.

Pay attention next time you watch a film to the way editing moves the plot along. Observe how shots are juxtaposed, how one leads to the next. Notice how movement is handled with editing. How long does each cut last, and how many go into a scene?

Filmmakers have been working with these ideas throughout the past century and have become quite accomplished in their use. So adept are they in the manipulation of these techniques that, for the most part, we don't even notice them at work. Now, as digital entertainment strides toward new creative horizons, we can make use of their experience to improve our own art.

If you want to add stunning animation sequences to your next title—and of course you do—here are the first two steps: look into a three-dimensional modeling/animation program, and start picking apart every movie you watch. Oh, and save me the aisle seat.

David Sieks is a contributing editor to Game Developer *and is absolutely no fun to go to the movies with. Contact him via e-mail at dsieks@arnarb.harvard.edu or through* Game Developer.