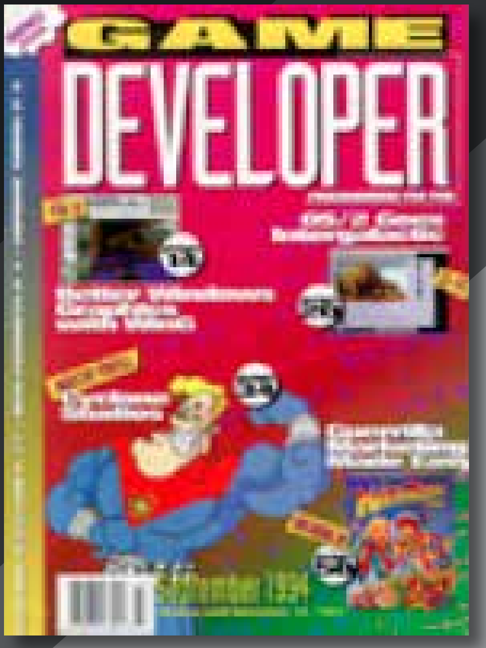
GAME DEVELOPER MAGAZINE

SEPTEMBER 1994



Back

Deflating the Ratings

Are you scared of the proposed video game rating system? Do you want to win the ratings game? Do you want to ignore the dictatorial commandments coming down from on high and put whatever you want in your game? Do you want your game to be available anywhere without censorship and still put in that disgusting scene that makes everybody wince? Here's what you have to do.

No legal shenanigans. No secret code words. No writing two versions. It's really simple—all you have to include are *choices*! Options are the whole key. If your players aren't forced to do violence, there shouldn't be a need to rate it in the game.

The Power of Money

One of the best examples of this tactic was put forth during one of the roundtable discussions at the most recent Computer Game Developer's Conference. The game in question was, as always, Doom. The idea was that if Id added another choice to the weapons category, any rating for violence the game received could be debated.

The weapon, or more correctly tool, to be added would be a wallet. Using the wallet, any creatures encountered could be bought off so they would not bother the player. While this might not make the game the most fun to play, it does remove the element of necessary violence.

Clearly this is a somewhat ridiculous solution and one I don't think Id should seriously consider, but the implications are clear. If players are offered a choice as to whether or not to commit the violent actions the ratings committees find so offensive, they will have to work twice as hard to justify whatever actions they are going to take. Committees and the public at large need to realize that rating interactive entertainment is not the same as rating a movie.

No Easy Answers

Obviously, this solution won't work for everyone. No amount of options is going to turn Mortal Kombat into *My Dinner with André*, but for some developers this approach should be considered. For the large project developer who has to sink a considerable amount of capital into a project, whose shape won't be determined until the later half of the development process, this approach makes sense.

And, while I'm not in favor of video game ratings, forcing creative minds to work out nonviolent alternatives is not the end of the world, nor the end of this industry.

The idea that video games should be rated by content has been around since the infamous Custer's Revenge on the Atari 2600, but developers today are still more or less free to do whatever they want. The backlash brought about by the latest round of violent video games won't last long, but could cause problems for all developers in the near future.

Ultimately it will be the channel, both retail and on-line, that will determine what games are made and sold in the future. And as sure as you can see naked butts on *NYPD Blue*, whatever restrictions are placed on video games today won't last over the long haul. ■

Alexander Antoniades
Associate Editor

Game Over!

We need your feedback! Send your cards, letters, and article suggestions to:

Game Developer
600 Harrison St., 4th Floor
San Francisco, CA 94107
Atten: Larry O'Brien

E-mail is even better:

GAME DEVELOPER

Editor Larry O'Brien
76702.705@compuserve.com
Associate Editor Alexander Antoniades
sander@mfi.com
Production Editors Barbara Hanscome
73611.633@compuserve.com
Nicole Claro
76702.1141@compuserve.com
Editorial Assistant Diane Anderson
diane_anderson@mfi.com
Editorial Intern Alison Long
406along@mfi.com

Cover Photography Charles Ingram Photography

Publisher Veronica Constanza
Project Coordinator Nicole Freeman
76702.706@compuserve.com
Group Director Regina Starr Ridley

Advertising Sales Staff
New England/Midwest
Angela Barnett (415) 905-4983
abarnett@mfi.com
West/Southwest
Yvonne Labat (415) 905-2353
ylabat@mfi.com

Marketing Manager Susan McDonald
Art Director/Marketing Christopher H. Clarke
Advertising Production Coordinator Denise Temple
Director of Production Andrew A. Mickus
Vice President/Circulation Jerry M. Okabe
Group Circulation Director Gina Oh
Circulation Manager Philip Payton *ppayton@mfi.com*
Newsstand Manager Pam Santoro
Reprints Krista Hiser (415) 905-2783

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

Back

A Mixed Bag

Nicole Claro
and Alex Dunne

How about the latest in computer graphics education, sound, and video drivers? And Alex Dunne updates the industry news with a look at the controversial idea of ratings for video games.

PRODUCTS

Videogame U.

Five years ago, I graduated from a small East Coast college known less for its form and structure than its lack thereof. If one more person asks if I majored in underwater basket weaving....Due to recent developments, I was lately musing over the fate of my alma mater when I learned of DigiPen Computer Graphics' newest foray into education. Lucky me! I can start all over at the DigiPen Applied Computer Graphics School, in Vancouver, B.C., Canada.

This fledgling institution offers a two-year program focused on the technological and engineering process of creating interactive multimedia programs. Of course, you must have some questions about their curriculum. And I just may have some answers.

- Is there a phys. ed. requirement?

Not that I know of. But there is a Foundation Year, during which students study algebra, algorithms, two- and three-dimensional transformations and volumes, and the basics of computer graphics.

- Is there ivy on the walls?

Well, I don't think so. But in the second year (called the Production Year), every student gets to develop his or her own game using Nintendo's Super NES Development System, which the company has generously supplied. The machines are attached to a regular Super NES and hooked into a PC, allowing students to program video games compatible for Nintendo's 16-bit cartridge system. During the Production Year, students also learn about storyboard presentation and final algorithms.

- What about SATs?

Here's what you need:

You must be a high school graduate and 18 or older. Consideration for acceptance is based on an entrance exam and evaluation by a screening committee, which reviews transcripts, letters of reference, and any applicable work experience.

- Do they offer a major in underwater basket weaving?

See my previous comments....

DigiPen Applied Computer Graphics School is registered with the Private Post-Secondary Education Commission of British Columbia and is considered an institute in the Canadian Educational System. In other words, it's legit. And I think it's the academic wave of the future.

For more information contact:
Jason Chu
DigiPen Applied
Computer Graphics School
530 Hornby St., 5th Fl.
Vancouver, B.C.
Canada V6C 2E7
Tel: (604) 682-0300

The Song is No Longer the Same

Here's my idea for a new conceptual art-rock band: tone-deaf singers, a rhythmically-challenged percussion section, and a guitarist who can't tell the difference between a major chord and a mike cord. Not only do none of them know much about music, everything they compose will be created in a matter of minutes. It does sound impossible.

But they'll all make beautiful music together—if one of them is an adventurous programmer. All you need is Arpeggio with TuneBuilder, the newest product from AirWorks Media. If you have a CD-ROM drive and a sound card, you're ready.

TuneBuilder is the main component of AirWorks' Arpeggio Self-Editing Music Library, which consists of 12 CDs that contain 335 selections in 25 different musical styles. But AirWorks' newest music editor will soon be available with many other major music libraries, including Killer Tracks and BMG.

You can apply Arpeggio to many different applications, ranging from business presentations to complex software projects. Just choose your music from Arpeggio, tell TuneBuilder how long it should be, click your mouse, and several edited versions will be created to your specifications. Using TuneBuilder, Arpeggio can create up to 200 different possiblities for every length chosen. If you want a more hands-on approach, you can still use TuneBuilder's high-speed cut editor capabilities.

Arpeggio with TuneBuilder features automatic self-editing; an intuitive mouse-driven graphic interface with powerful cut, paste, and play features; direct play CD-ROM functions; and Redbook Audio for processing music. Powerful search capabilities help find user-editable text descriptions, musical fee, tempo marking, beats per minute, and style. It runs on DOS, Macintosh, Windows, and Amiga, and supports .AFC, .AIF, .AU, .RAW, .SMP, .SND, .VOC, and .WAV file formats.

The Arpeggio Library is divided into four categories—Jingles and Themes, Upbeat and Contemporary, Moods and Background, and Business and Presentations. You can purchase Arpeggio as a full library, one or more categories, or one or more single volumes. Two license options are available, one for broadcast or commercial use and one for in-house corporate use. Price varies according to version and license.

Of course, my art-rock band will only become a reality if it exists within the confines of a computer application...but what could be more conceptual than that?

For more information contact:
AirWorks Media Ltd.
1100 Woodward Ave., Ste. 120
Bloomfield Hills, Mich. 48304
Tel: (800) 525-5962 or
(810) 645-5730

Render It in 3D

Let's say I have an idea for a game I want to develop. Maybe I've just gotten my degree from, oh, I don't know, a video game programming institution in Vancouver, B.C. Now, let's say I don't have the money for top-of-the-line royalties. In fact, all I really want to do is write a program that can run on any PC. What's the three-dimensional API for me?

The search is over! Criterion Software Ltd.'s RenderWare is the first interactive three-dimensional graphics API for Windows. RenderWare is designed to run on low-cost PCs and provides impressive graphics without the need for special three-dimensional graphics accelerators. But RenderWare isn't just for the hobbyist. High on functionality, RenderWare is a device-independent three-dimensional graphics API consisting of a minimum number of object types coupled with a full set of associated functions, including advanced shading and texturing. The API is totally software based, and its performance increases as processor performance increases.

You can apply RenderWare to almost any project including multimedia, visual simulation, scientific visualization, CAD, virtual reality, presentation graphics, entertainment and games, and education and training. Priced from $10,000, the RenderWare software development kit includes development library, debugging library, documentation, examples, and demos. RenderWare requires Windows 3.1 running on a 386/SX or better with 4MB of RAM.

For more information contact:
Criterion Software Ltd.
17-20 Frederick Sanger Rd.
Guildford, Surrey
GU2 5YD, U.K.
Tel: 011 44 483 448800
Fax: 011 44 483 574360

Video Drivers in DOS

Tenberry Systems Inc., formerly Rational Systems, now provides DOS support for Intel's Indeo video compression and decompression software. Indeo video is a software technology that enables software-only video playback whether you develop games, graphic animation, or multimedia applications. As long as you're working with any i486 or Pentium processor systems, no additional hardware is needed. This new system lets developers decompress Indeo video compressed data under DOS using Intel's Indeo video driver for Windows. The concept is based on Tenberry's DOS/4G, a 32-bit DOS extender for systems and application programming.

For more information contact:
Tenberry Systems Inc.
220 N. Main St., 2nd Fl.
Natick, Mass. 01760
Tel: (508) 653-6006

Nicole Claro is departments editor for Software Development *magazine.*

INDUSTRY NEWS

The Ratings Game

An issue that traces its roots to the movie and music industries has finally appeared in software entertainment: ratings. I'm not talking about a Siskel and Ebert "thumbs up" for enjoyment value. Groups are pushing for cautionary notices on game packages—the same type of warning that Tipper Gore backed for music packaging.

Spurred by the increasing amount of blood, sex, and profanity in games, lawmakers and industry organizations are starting to question this material in video and computer games. "We rate movies and restrict their viewing to adults," the line of thinking goes, "so why allow scenes of carnage, nudity, and profanity to go unchecked in software entertainment?" The issue has led to the creation of two factions, each with plans to inform consumers about the content of games prior to purchase. Interestingly, in the game rating issue (unlike the brouhaha stirred up by music warning labels), both sides of the rating debate are publishing their own rating specs: nobody appears to be taking a stand against game ratings.

Shot Heard 'Round The World: The Lantos Bill

The first volley in the ratings dispute was fired by U.S. Congressman Tom Lantos (D-California), who introduced a bill to Congress on February 3rd of this year called the Video Game Rating Act of 1994. The purpose of this bill is to "provide parents with information about the nature of video games which are used in homes or public areas, including arcades or family entertainment centers." The bill would establish a five-member commission, appointed by the President, to draw together a plan for a voluntary ratings system. In a section entitled "Regulatory Authority," the bill contains language that indicates the possibility of further mandatory steps: "...the Commission may promulgate regulations requiring manufacturers and sellers of video games to provide adequate information relating to violence or sexually explicit content of such video games to purchasers and users."

In response, video game manufacturers, feeling the sting of public and legislative criticism about game content, have banded together to form the Interactive Digital Software Association (IDSA). The group is made up of a dozen or so manufacturers, including cartridge behemoths like Sega, Nintendo, Electronic Arts, Atari, and Acclaim. The IDSA has proposed its own commission to assign game ratings to video games and computer games. The IDSA plan, however, would require a fee to cover the costs (estimates range from $300 to $500) of the rating process. Most likely, that fee would be passed along to the game publishers.

Although the IDSA's plan has been hailed by Lantos, it has been derided by others as a poor solution that could affect the viability of small game publishers unable to afford this fee. To fight the IDSA's concept, another coalition comprising the Software Publisher's Association (SPA), the Shareware Trade Association and Resources (STAR), the Educational Software Cooperative (ESC), and the Association of Shareware Authors and Distributors (ASAD) has formed. This alliance, taken as a whole, represents over 3,000 software publishers.

STAR, in a response to the IDSA plan, states that the plan "...places a burdensome and unnecessary expense on small authors and publishers." This raises a question: In the event many startups couldn't afford the fee, would the IDSA be willing to subsidize their fee (which, taken together, could amount to millions of dollars)? The SPA/STAR/ESC/ASAD alliance also expresses the fear of centralizing so much power over ratings in a single body, as IDSA's plan would. The alliance backs a "content disclosure system" that puts the burden of labeling game packaging in the domain of the developer or publisher, "...who would have a much greater understanding of the presentation and content of a game than a reviewer could get from a video or storyboards." The system would be self-policing, on the assumption that developers and publishers would not want to mis-rate a game and risk a fallout with either the public or their distributor.

At the heart of this issue—and what I find most fascinating—is the concept of ratings. The goal of game ratings, regardless of the group proposing the system, has been to target three categories of offensive material within games: nudity, profanity, and violence.

Of these three evils, violence appears to be the most difficult to define, probably due to the fact that established guidelines are already in place for nudity and profanity in other forms of media. The debates within the industry attempting to identify and categorize it have fallen somewhere between an undergraduate philosophy discussion and a *Saturday Night Live* skit. Is a *Tom & Jerry* cartoon considered violent? If so, to what degree?

Six Levels of Brutality

In response to this dilemma, several people have suggested criteria with which to rate game violence—criteria that would leave as little to a reviewer's interpretation as possible. Using these rules, game censors could establish to what degree any game is violent, without their own values clouding the game's rating. One interesting classification scheme (a "violence-meter" I suppose) proposed by someone on CompuServe divided vio-

lence into the following six categories:
1. Shooting objects
2. Killing unreal life forms, without blood and guts spraying out
3. Killing unreal life forms, with blood and guts spraying out
4. Killing humans, without blood and guts spraying out
5. Killing humans, with blood and guts spraying out
6. Player rewarded for killing, torturing, or goring innocent humans.

I know of quite a few Bugs Bunny episodes that would fall into category six. On the other hand, so would Mortal Kombat, which has some finishing moves that would turn a trauma nurse green. A case can be made for the "Oh, it's only fun and games" faction and the "This is damaging the values of our youth" faction.

Unfortunately, Mortal Kombat is an easy target (as are Doom and Wolfenstein), because it practically flaunts its violence, and the killings are so...ahem...unique that they have raised it to a class unto itself. But other games may not be so easily classified. Battle Chess has some graphic death scenes, and in Populous you're trying to annihilate an entire tribe of people. I don't find either harmful or particularly offensive, yet each would receive a rating of 4 or higher in the suggested rating scheme I've detailed. An astute person raised the question of how many human-like qualities qualifies a character as human? Is killing a hobbit (which is somewhat human in appearance) enough to garner a 4 rating? How about an elf, dwarf, a cyborg with a human-like exterior, Frankenstein...you probably get the picture. Even the most nonpartisan reviewer in the world would have a difficult time grappling with a gnome's anatomy to try to classify it as a human or not.

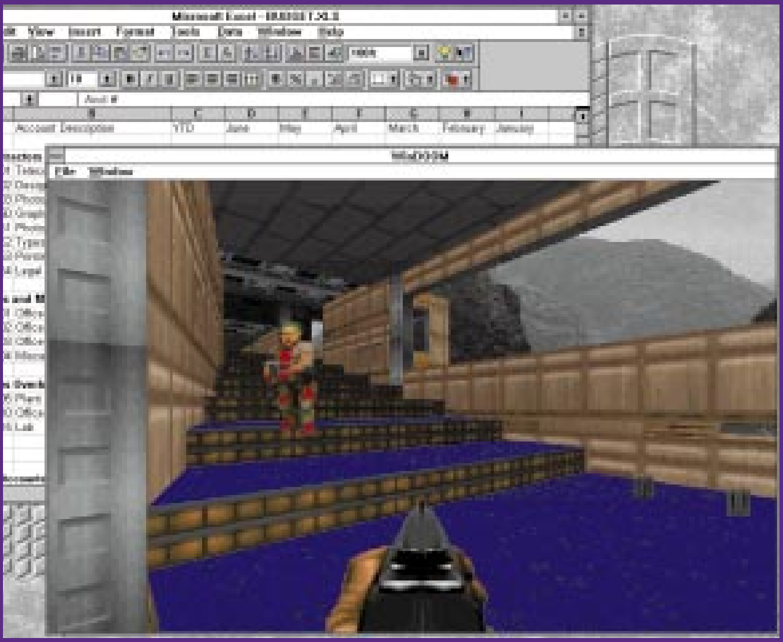
A sidelight to the ratings controversy is that all of the proposed schemes (with the possible exception of Lantos' bill) would be implemented on a voluntary basis. Developers and publishers wouldn't be mandated to have their games rated...as long as they didn't want to sell unrated games through some of the largest consumer retail chains. Already, rumors are spreading that Toys R Us and K-Mart would not carry unrated games. Didn't that almost happen to Spinal Tap's album, *Smell The Glove*?

The subject of ratings, like many questions of what is morally acceptable, draws people into its vortex. Whether it's book burnings or warning labels on compact discs, there are plenty of fervent people who will argue their side of the case. Unlike book burnings and music warnings, though, the fate of games appears to be sealed, as no champion against game ratings has stepped forth. Perhaps in our "politically correct" world today, we are no longer willing to fight the trend toward having someone else taste-test everything for us. ■

Alex Dunne is product review editor for Software Development *magazine.*

Back

A Whirlwind Tour of WinG



When WinDoom was ported, the rendering vs. stretching issue was left to the user by providing a menu of choices. Now you can set it to render to the current window size or preset the size and stretch to the current window size.

If you're like me, the first time you saw Microsoft Windows 3.0 and its program manager, you went straight for the Games program group. Like me, you probably expected to find a game as different from DOS games as Windows is different from DOS itself. Instead, you found Solitaire. Not a bad version of Solitaire, but Solitaire nonetheless. If you waited until 3.1 to check out Windows, you also found Minesweeper—a bit more exciting, but you wouldn't call it "high-performance."

Expectations for Windows games have been very low. When Microsoft released a set of games called Arcade last year, reviewers were shocked. They couldn't believe games of Arcade's quality could be done on Windows. Arcade is a great set of games, but we are talking about 1970s technology on 1990s computers! Their enthusiasm was unfounded: Arcade is nothing compared to the games you find on DOS. A Pentium probably has more on-chip cache than the original Asteroids game had main memory.

Sure, operating systems of today do more than they did back then (did they even have operating systems back then?), and I can play Asteroids while simultaneously running other applications on the same desktop, but is this all we can expect from our brand new machines running Windows? On the same hardware, DOS games have consistently pushed the performance envelope with the current crop doing full-screen texture-mapped worlds at 30 frames per second. What's the crucial difference between DOS games and Windows games? Graphics performance.

Finally there's help: WinG. WinG is a library that eliminates the performance difference between DOS and Windows graphics, giving Windows games graphics performance at or above their DOS counterparts on the same hardware.

Current Windows Graphics—Slow?

We're interested in raw blt (bit level transfer) performance: transferring pixels to the screen in blocks. Most high-performance games try to achieve smooth animation by hiding the rendering and only allow the player to see the resulting frame. These games compose images into buffers, then quickly update the display. While the composition phase is usually application-by Chris Hecker

Does graphics performance set DOS and Windows a world apart? Think again. Because of WinG, Window's graphics are flying high, giving performance at or above their DOS counterparts.

specific (each game renders using its own special algorithms), only a few popular techniques for updating the display exist.

Update techniques fall into two groups: blting and page flipping. The trade-offs between the two techniques on current PC hardware are far too complex to cover here, but suffice it to say that high-performance DOS games use both techniques (for example, System Shock and Ultima Underworld I and II blt, while Doom page flips). It is fairly easy to move a game from blting to page flipping or vice versa.

Windows does not currently allow page flipping, so we will deal with blt performance. Although we've said graphics speed (or lack thereof) is the major impediment to high-performance Windows games, if you time the `BitBlt` function, you will find the bandwidth comparable to what you find under DOS for the same resolutions. The catch is `BitBlt` transfers pixels from objects called `HBITMAP`s, not from memory the application owns.

Applications are not allowed to touch the bits of an `HBITMAP` directly, they must use Windows Graphics Device Interface (GDI) functions, like `LineTo`, `SetPixel`, and `Rectangle`. GDI provides a rich set of two-dimensional graphics functions that are perfect for applications like spreadsheets and word processors, but you will not find a `TextureMapPolygon` function anywhere in the Windows API documentation. For this reason, games need to render directly to memory, and GDI does not allow them this luxury with `HBITMAP`s.

Windows does provide objects called Device Independent Bitmaps (DIBs), which applications can access directly, but the APIs for transferring DIBs to the screen (`StretchDIBits` and `SetDIBitsToDevice`) are typically three to 20 times slower than `BitBlt` and therefore not competitive with DOS blt bandwidth.

WinGBitmaps—a Hybrid

WinG introduces a new kind of object: the `WinGBitmap`. `WinGBitmap`s are both DIBs and `HBITMAP`s. Applications get a pointer to the bits like a DIB, and like an `HBITMAP`, WinG will transfer them to the screen quickly. How quickly? At the 1994 Game Developer's Conference, we demonstrated a Windows version of Doom, WinDoom, running at about the same speed under Windows as the DOS version on the same hardware. Better yet, it only took a weekend to do the port.

Porting a DOS Game to WinG

I don't have space in this article to develop a DOS game and then port it to Windows and WinG, but I will describe a typical DOS game's architecture and discuss how to move it to WinG. Let's assume our game has five major parts:

- Setup
- Get input events
- Run the simulation
- Render into a buffer
- Blt the buffer to the screen.

During `Setup`, the program allocates the off-screen buffer, creates the palette, and initializes the simulator. Next, it gets any user input and uses that information to run the simulator for a single time slice. The results of the simulation are rendered into a buffer, and the buffer is blted to the screen. We're ignoring synchronization, sound, networking, user interface, and whatnot, but you get the idea.

Under Windows, the setup phase needs to initialize Windows-specific elements, like the application window, but most of the setup code stays the same. One interesting difference is that, unlike the DOS version where your application allocates the buffer memory, you must call `WinGCreateBitmap` with `BITMAPINFO` (a structure describing the size and format of the `WinGBitmap`) to allocate the buffer, and WinG will return the memory pointer. The application uses this pointer to draw on the `WinGBitmap` surface directly.

The application will also need to use GDI palette APIs to create and realize the game's palette. GDI realizes a palette when it copies the description of the palette colors into the video hardware. Because multiple applications can share the hardware palette, this can get a bit tricky, but there is plenty of palette sample code in the WinG development kit to illuminate matters.

User input is slightly more difficult. Well-behaved Windows applications must yield control to the system fairly often in case the user wants to switch away to another application. Normal applications like word processors call the `GetMessage` API to process their user input messages. If there are no messages for the application, `GetMessage` doesn't return until one comes in.

A game can't use `GetMessage` because even if the player isn't providing input to the application, the simulation must still run. You don't want the whole game to stop when the user stops pushing keys or moving the mouse, so Windows provides an API called `PeekMessage`. This API returns immediately even if there are no messages so the game can continue the simulation. The subtleties of `PeekMessage` in particular and event-driven architectures in general are beyond the scope of this article, but I will provide you with an appropriate reference.

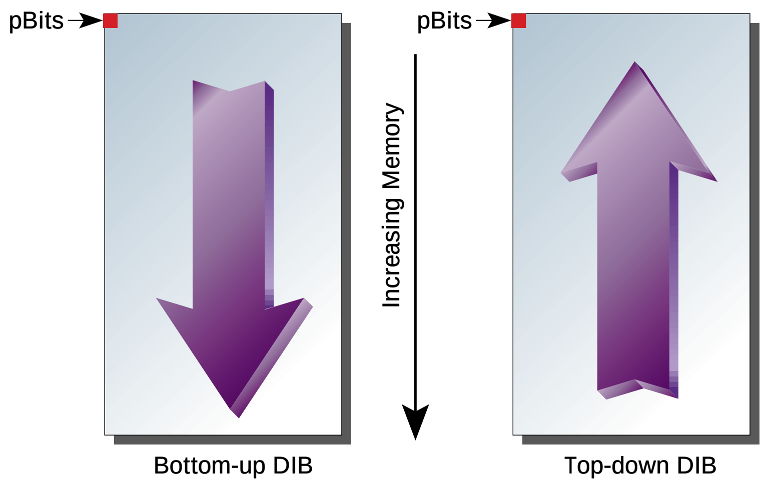
The game simulation code should work unchanged on Windows. Once the user input is translated from Windows messages to the application-specific format, the simulation should run normally.

Your game's rendering code should also work unchanged. The only caveat is that `WinGBitmap` scanlines are `dword` aligned, so if for some reason you need a 201-wide bitmap, you'll need to know the start of the next scanline is actually 204 bytes from the current scanline, not 201 bytes.

Once composition is complete, you blt the buffer to the screen with `WinGBitBlt` or `WinGStretchBlt`. As its name implies, `WinGStretchBlt` will stretch or compress the `WinGBitmap` as it blts, where `WinGBitBlt` simply transfers the `WinGBitmap` to the screen.

Once you have your game running on Windows, it's time to make it run fast. You'll also want to take advantage of the benefits of running in a windowed environment, so we'll talk about some of those issues as well.

Figure 1. DIB Orientations



Setup

Our naive port called `WinGCreateBitmap` with the description of the `WinGBitmap` we wanted. To achieve maximum blt performance during the screen update phase, we'll ask WinG for a little help during our optimized setup. Although WinG is fast under almost any circumstances, there will always be a particular `WinGBitmap` format that is the absolute fastest to blt on the current display, and the `WinGRecommendDIBFormat` API will tell us what that format is at run time.

The most important difference between the DIB formats that we'll get back from `WinGRecommendDIBFormat` is the DIB orientation. There are two DIB orientations: bottom-up and top-down, illustrated in Figure 1. Both kinds of DIBs consist of a `BITMAPINFO` structure and a pointer to the bits. The `BITMAPINFO` contains information such as the width, height, number of bits per pixel, and the color table of the DIB. For bottom-up DIBs, the bits pointer points to the bottom-most scanline in the DIB.

Increasing memory addresses means going up the DIB image, hence the term bottom-up. This is probably the exact opposite of the memory bitmaps you've dealt with before and is the opposite of most video displays (notably mode 13h VGA, for example). Top-down DIBs are more familiar: the bits pointer points to the top-most scanline, and increasing memory addresses go down the image. Life gets interesting because WinG might recommend either DIB format at runtime, and high-performance games should be able to deal with both. This isn't as hard as it sounds. I'll go over the details in the section on rendering.

Once we have the recommended DIB format, we pass the information to `WinGCreateBitmap` and go on to our palette setup. For optimal performance, a WinG application should have an "identity palette mapping." An identity palette mapping means the color table in the `WinGBitmap` and the palette in the display hardware match exactly. In this case, WinG can block-transfer the pixels in the `WinGBitmap` to the screen without translating them. If the palette mapping is not identity, WinG needs to translate each pixel as it is blted, which is slow. We'll cover this briefly, and if you still don't get it, there is plenty of excruciatingly detailed documentation and sample code in the WinG development kit.

Windows runs multiple applications at the same time. There is only one hardware palette—something has to give. The compromise is that each application requests the hardware palette (called the system palette) by calling `RealizePalette`. Windows may or may not let the application have the entire system palette depending on a number of factors, like whether the application is in the foreground, whether there are other palette applications around, and so on.

Even if Windows does give the application the system palette, the system tries to minimize the palette entries used by each application by collapsing any duplicate colors into the first instance of that color. In addition, each `WinGBitmap` has an application-defined color table associated with it, and the color table must match the system palette for the mapping to be identity. If all this sounds complicated, it is, but once you understand it, you'll be able to charge outlandish consulting fees to other game developers, so it's worth your time to learn. Besides, your blts will go from mediocre to blazing once you get an identity palette mapping.

WinG can help in your quest for an identity mapping by spitting out debugging information. You can set two flags in the win.ini configuration file to direct WinG to tell you what is going on. The `Debug` flag makes WinG tell you if you have an identity palette mapping, and the `DebugPalette` flag makes it tell you how each color table index in your `WinGBitmap` maps to the current system palette if that mapping is not identity. So, if you can't figure out why you don't have an identity palette, you can turn on `DebugPalette` and see messages like:

```
WinG: Palette mapping is not identity.
WinG: Color table index 123 maps to
      system palette entry 5.
```

You can take this information and see exactly why you aren't getting an identity mapping.

As soon as you've figured out the intricacies of identity palettes, you'll need to make a user interface decision: `SYSPAL_STATIC` mode or `SYSPAL_NOSTATIC` mode. Windows normally reserves 20 colors in the system palette and does not let applications overwrite them. This keeps a single palette application from making all other applications look horrible—other applications always have at least those 20 colors, called the static colors, to map to, even if an application realizes an all-black palette. As with most things in Windows, there's a way around the static colors: `SetSystemPaletteUse`. If you call `SetSystemPaletteUse` with `SYSPAL_NOSTATIC`, Windows will let you overwrite 18 of the 20 static colors, leaving only black at entry 0 and white at entry 255.

`SYSPAL_NOSTATIC` applications make the Windows desktop look gross, while `SYSPAL_STATIC` applications only get 236 colors out of a possible 256. You'll need to choose which mode to use as you develop your game. It is possible to use SYSPAL_NOSTATIC when you have a maximized window (users won't be able to see the off-colored desktop anyway) and SYSPAL_STATIC when you're windowed (and users can see the program manager and other applications), but your game must do the extra work.

Rendering

High-performance games have optimized rendering algorithms, and most of this code can be left alone, although your rendering code will need to deal with top-down and bottom-up DIBs for best performance. The impact this has on most rendering code is minimal. When you step from scanline to scanline, you need to use a signed number. For example, let's say this is your rendering loop for a 320 byte wide buffer:

```
   ; edi points to destination scanline
   mov     edi,pBits
loop_top:
   ; draw some pixels
   mov     [edi],ThisValue
   mov     [edi+4],ThatValue
   mov     [edi+8],TheOtherValue
   add     edi,320    ; point to ext
                        scanline
   dec     ScansLeft ; if we´re not done,
   jnz     loop_top  ; do it again
```

Although simple, this type of loop is the core of most scanline renderers. After changing two lines, this code can handle both DIB orientations at run time:

```
mov   edi,pBits
```

becomes:

```
mov   edi,pTopScanline
```

where pTopScanline is the first scanline (pBits) on top-down DIBs and the last scanline (pBits + WidthInBytes * (Height - 1)) on bottom-up DIBs. The second change is:

```
add   edi, 320
```

to:

```
add   edi,DeltaScan
```

where DeltaScan is 320 for top-down DIBs and -320 for bottom-up DIBs. This change causes the renderer to always move down the image, increasing edi for top-down and decreasing it for bottom-up.

A second issue affecting the renderer is variable-sized viewports. Because Windows runs at whatever resolution the user chooses, games should be able to handle different window sizes. There are two ways to do this: the game can render at different resolutions, or the game can use WinGStretchBlt to stretch a constant-sized buffer to the variable-sized window. The former is a rendering issue, the latter affects the blt/update code as well.

Blting

There are tradeoffs between rendering at the viewport resolution and calling WinGBitBlt to blt the buffer and rendering at a lower resolution and calling WinGStretchBlt to expand the buffer to the viewport resolution. If your renderer can handle high-resolution buffers, you'll get the best looking results by rendering at the resolution of the viewport, but you might find the performance is too slow. If your game is pixel-bound, like Doom (in other words, it spends more time rendering a pixel than WinG spends blting or stretching that pixel), you may want to take advantage of the high-performance stretch code in WinGStretchBlt, render to a low resolution buffer, and stretch it to fill the viewport.

When we ported WinDoom, we left the rendering vs. stretching issue to the user by providing a menu of choices. You can set it to render to the current window size (which really slowed down as the window got larger), or it could render at a preset size and stretch to the current window size. WinGStretchBlt is extremely fast, so the stretching option usually resulted in the best frame rate, but it didn't look as nice as the full rendered version. Most DOS games have level-of-detail settings, so users can choose stretching versus rendering as they like.

Other Issues

It's been said that the best and worst thing about Windows is that it runs on an incredible variety of hardware. To make the most of this variety, your game will need to configure itself to the run-time platform, like WinG does at startup with the display performance test. Is it faster to stretch or render? The answer will change depending on the user's hardware and software configuration, so be prepared. Is it faster to update dirty rectangles or blt the whole buffer? Again, this can change from machine to machine. Time it and you'll never go wrong.

This has been a whirlwind tour of WinG game development, but we've touched on the major issues. Once you are seriously into Windows programming, get the WinG development kit for yourself and play with the sample applications to get first-hand experience, then port your game to Windows in no time flat. ■

Chris Hecker works for a large software company in the Pacific Northwest. He can't mention the name because then he'll need all sorts of disclaimers. It's just a coincidence that he can be reached at checker@microsoft.com. or through Game Developer *magazine.*

WINGING IT

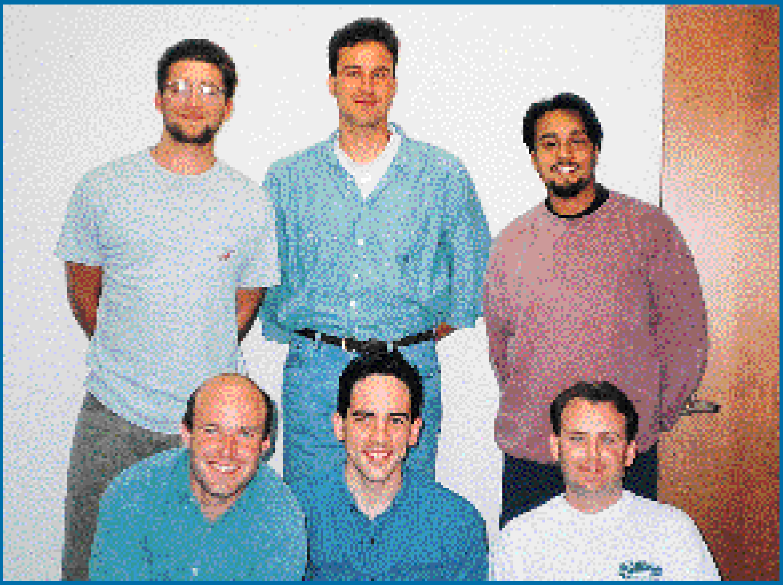
Is it time you port your own game to Windows? Then it's time to start using WinG. For the satisfaction of porting your own game in no time flat, consult these resources. Don't sit around while someone does it for you—WinG it!

FOR BEGINNERS:
Programming Windows 3.1 by Charles Petzold (Microsoft Press, 1992)

FOR EXPERIENCED DEVELOPERS:
The Microsoft Developer's Network CD-ROM (Microsoft Developer Network, (800) 759-5474).

FOR EVERYONE:
Microsoft's WinG Development Kit is available from the winmm forum on CompuServe or on ftp.microsoft.com

Industry Profile: Cyclone Studios



From rags to riches, this startup company went from a two-person bedroom operation to a six-person team, complete with office space, cool logo, and (hopefully) great games. The Cyclone Studios team is (standing) Heli, Maarten, Subha, (kneeling) Helmut, Ron, and Greg.

It all started in late 1993 when, after months of casually talking about it, a close friend and I decided to leave our well-paying, perfectly secure and respectable jobs and take the plunge into independent video game development. We—that is, myself and Ron Little, a talented programmer I had known since college—decided to call ourselves Cyclone Studios and made it our mission to build original, top-notch games for next-generation systems like the 3DO Interactive Multiplayer. We wanted Cyclone to become synonymous with high-energy, fun, quality entertainment—the equivalent of Steven Spielberg's Amblin Entertainment in the video game world.

Of course, there were a couple of minor obstacles in our way. First, neither Ron nor I had ever built a video game company before. Second, we had virtually no money to do it. We started with a very simple, straightforward plan: We would begin developing an original game for the 3DO system and, as soon as possible, present a prototype to either game publishers or other investors who would give us the money needed to finish the project.

Since we didn't have a lot of money between us, we'd work out of our homes and push as hard as we could for the next six months, which is when our savings accounts—that is, life support systems—would run dry. Fortunately, by the end of that period, we had completed our game engine, a preliminary script, and polished up a presentation for publishers. As it happened, our game, which we'd based on a proven, profitable genre and featured a strong, marketable character property, was picked up almost immediately by 3DO's own publishing arm, Studio 3DO.

And what a relief that was, since, with 3DO's cash advances, we could finally afford to grow Cyclone into a genuine company. First, we could hire our first employee, Greg, a top-notch artist who'd do much of the artwork for our first game. We could finally move out of our bedrooms and into some office space. Most importantly, with a deal from a publisher like 3DO, we were able to convince a private investor to initially fund two other games we'd wanted to try. That meant hiring Heli and Subha, two more capable game programmers who brought Cyclone's staffing up to five peo-

by Helmut Kobler

Started on a shoestring, Cyclone Studios is taking its market by storm. By concentrating its efforts on one platform—3DO—this amazing startup is on its way to becoming a major industry player.

ple, with one title in full production and another two in the early stages.

Birth of a Whirlwind

All of a sudden, Cyclone grew from being a couple of people working out of their bedrooms to a small but definitely real game development shop. That's what this article is about. The game industry is a dynamic place; there are lots of talented people out there, and I suspect that many are already involved in small, young game development shops like Cyclone or at least harbor secret fantasies of breaking off from the established companies they work for and striking out on their own.

I wanted to write an article about Cyclone's own experiences in hopes of giving aspiring game makers out there some sort of useful perspective on going into independent game development—for instance, how to pick a hardware platform to focus your efforts on; how you can hire great people into your company even though you may not be able to pay top dollar just yet; and how to fund your initial growth without giving away too much equity in your fledgling company.

I'm by no means suggesting that Cyclone Studios' own approach and solutions to these issues are the only ones or even the right ones (time will tell that). Right or wrong, though, I think our experiences will give you greater insight into the challenges of getting a game start-up off the ground. So enjoy!

Structuring the Company

One of the most important and fundamental decisions a new game developer has to make is whether to stick to just making games or to also take on marketing and distribution, too—in other words, become a full-fledged publisher. When we started Cyclone Studios, we thought self-publishing our first game would be best, instead of looking for an existing publisher to pick it up. It's the publisher, after all, that makes most of the profits from a popular title. Publishers are also free to strike deals with talented development shops, so they can pick up hot games they didn't actually have to create from scratch.

We certainly liked those benefits, but after a few months, we also came to the conclusion that simply developing our game was a full-blown undertaking, and financing just the production process—all the animation, programming, and other art, the cinematic sequences, the musical scores, the voice actors, the script writers, and so on—was definitely beyond our means. Trying to add the burdens of manufacturing, marketing, and distribution would have been totally insane. (If we'd been writing games for computers, rather than game consoles, self-publishing might have been more workable; for instance, in the PC world, we could have released our game as shareware and taken a grassroots approach to marketing and distribution.)

Shareware or not, though, doing a really competent job of publishing a video game these days is way beyond the means of most game shops around. It's all a question of money. To do decent advertising, you need lots of money. To keep your shelf space secure, you need lots of money. To woo the all-important press, you need more money. And to stage the world-wide event marketing campaigns that publishers like Acclaim have become so good at—the TV ad blitz, the movie trailers, the merchandising tie-ins—you practically need a mint.

If you've got a venture capitalist or some other corporate sponsor who's offering you millions of dollars, then okay, you've probably got the money to compete at this kind of level. If not, though, trying to publish your own games in such a competitive atmosphere seems like an incredibly heavy and risky burden to bear. In fact, even some of today's small- and medium-sized publishers may face rough days ahead because, as with all industries that begin to mature, I think video games will come to be dominated by a handful of giant studios. And those publishers that don't grow into big league proportions may end up getting squashed.

Should small, independent game developers be worried about their fate? I don't think so. No matter how strong and dominant the industry's existing publishers become, they're still always going to need great games, and great games aren't something a publisher can simply churn out of its internal production department. Like movie studios, good game publishers draw on lots of outside production talent to get the best possible product.

All a small developer like Cyclone has to do is focus on coming up with some of the most exciting and engaging ideas possible and maintain a lean, competent staff that's great with execution. If a developer can do this, it's possible for even a tiny company to ally itself with top-tier publishers and dip into some of the deepest, richest pockets in the industry.

The Platform

When we decided to start Cyclone Studios, one of the first and most important issues we had to consider was the hardware platform—Macintosh, PC, Sega, Nintendo, or 3DO—that we'd develop our games for. For many established developers, this isn't even an issue, since they have the money and people power to develop for a range of formats. But for smaller companies like Cyclone, we thought it better to pick a platform we could develop real expertise for and continually build on as we started future games.

Of course, there were plenty of factors to weigh before choosing a preferred platform. For instance, if we were going to develop for video game consoles rather than PCs, we'd have to purchase expensive development systems from Sega, Nintendo, 3DO, or whoever else. And there was always the question of whether a platform would have the mature tools to make life livable during development.

The most important question we asked ourselves was whether or not to develop for new, untested platforms like the 3DO, Sega Saturn, Sony PSX, or Atari Jaguar. New systems like these never have the huge installed bases that make developers' mouths water, but they offer less competition for first-generation developers and a chance for a small development shop to make an early name for itself on an up-and-coming system.

It was this "new world" factor that pushed us toward 3DO, since the system represented such a new and untapped market for developers. When we made our decision, existing platforms like the Sega, Nintendo, and the PC, were already well established, and had more than their fair share of experienced, entrenched developers.

In this kind of environment, we felt we'd have a tough time convincing a game publisher to back the projects of a totally unknown, untried company, when there was already a whole stable of tried-and-true developers to choose from. 3DO, on the other hand, was virgin territory, with many of the industry's established developers waiting by the sidelines until 3DO became a clear success.

Being small and entrepreneurial, however, we didn't mind diving right in with the idea that 3DO and other publishers backing the system would be eager to buy any quality games in-progress once the installed base began to grow. The fact that the industry's bigger, richer developers were not giving 3DO their full attention meant we'd have a better chance of selling our games.

You should understand that I'm not trying to sell 3DO specifically as the land of opportunity for small game developers. I'm only using Cyclone Studios' own experience to show the advantages of picking a new platform—3DO, Jaguar, Sony PSX, or whatever—where it's easier for a small developer to make a difference and where you'll be well positioned if that platform begins to take off. Of course, if the platform you bet on doesn't do so well, your company will have thrown a lot of precious time and money down the tubes. On the other hand, if a small, hungry company can't stand taking a few risks, then who can?

Other than being virgin territory, there was a final factor that made betting on a new platform like 3DO worthwhile to us: ease of development. 3DO—and I suspect other next-generation systems like the Saturn and Sony—is a pleasure to develop for. In 3DO's specific case, there's a lot of custom hardware and internal libraries that make doing efficient animation, sound effects, background music, and full-motion video fairly easy on our programmers.

Since the overall hardware is fairly robust, most of our development is done in C or C++, with patches of RISC assembly language for strategic kicks. The result is that we can have a game engine up and running in just a few months (depending on the genre, of course), which means we spend less money getting a game's fundamentals to the point where we can present it to a publisher. Had we chosen a PC or a 16-bit Nintendo or Sega as Cyclone's target platform, our development cycles would certainly be slower and more expensive.

The Funding

After the euphoria of starting our own game company wore off, our next immediate concern was how to pay the initial development costs for the first game we'd be working on. Had we had a long, distinguished track record as game developers, I think we could have spent a couple of weeks writing up an initial game idea and stood a reasonable chance of finding a publisher to advance us all the production money we'd need.

Since Cyclone's founders didn't have specific, previously published games to point to, however, that option seemed less likely. Even if we had a worthy track record, there was still the chance that we wouldn't find a publisher willing to back a totally undeveloped idea. We'd have to shoulder the initial development costs ourselves, dedicating a few months to preparing our game to the point where we could interest a publisher.

I suspect that lots of aspiring game developers will have to do the same thing for a while. If so, your success comes down largely to a question of financial capacity—in other words, will you have the savings, loans from friends or family, or whatever other financial means to support yourself and your team through a game's initial development. If you can't be reasonably sure of this ability, I wouldn't even go forward, since it's really counterproductive to develop a game in fits and starts because you keep running out of money.

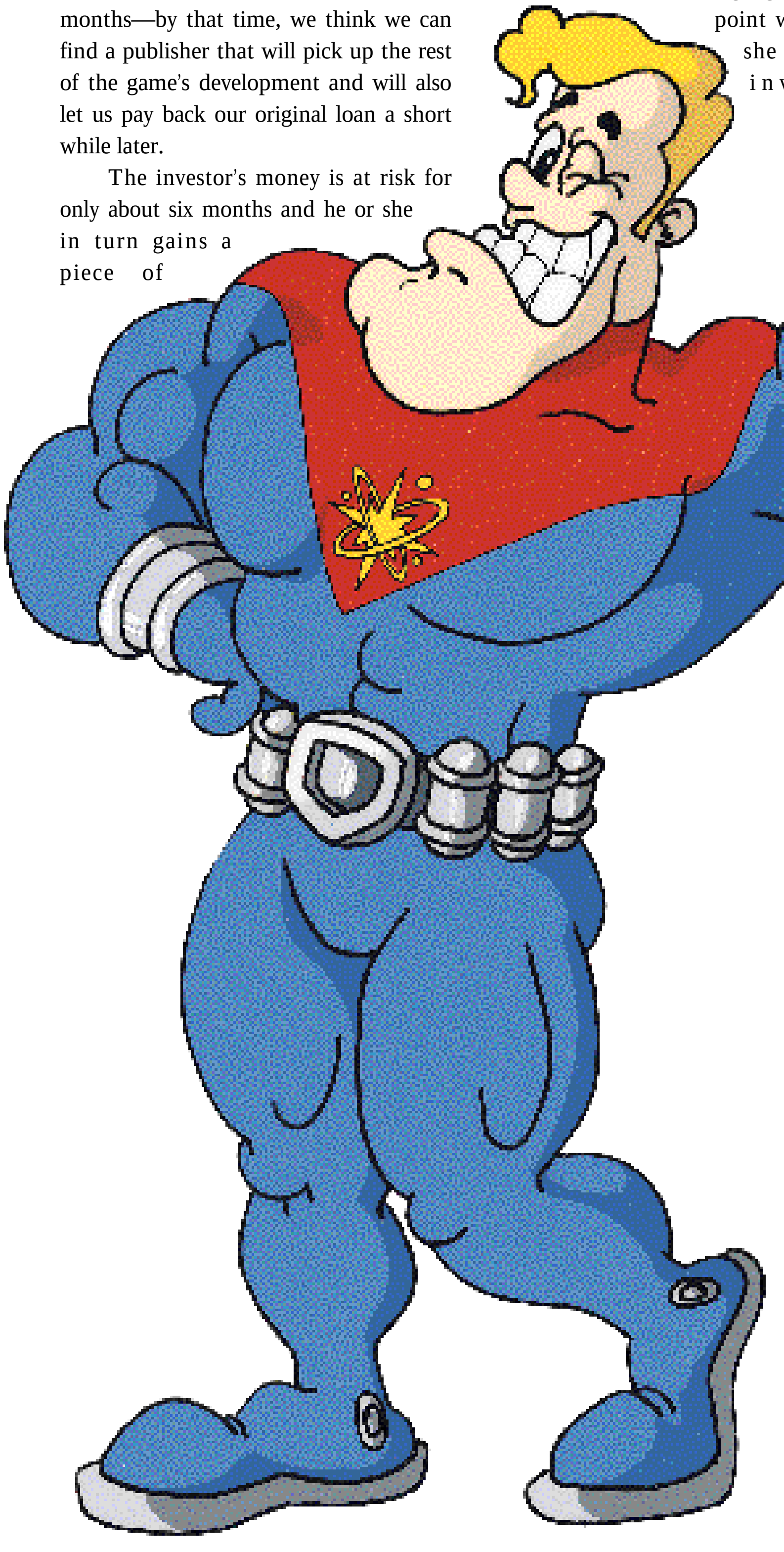
Fortunately for Cyclone Studios (and its founders' respective bank accounts) those lean days of living on Campbell's Soup selections and working out of our bedrooms came to an end after about five months of development, when we struck a publishing deal for our first game and the advance checks began coming in. Unfortunately, our welcomed advance money only covered work on our first game, while we were ready to start two other games we'd recently come up with.

Small, growing companies face this issue all the time; as soon as one source of money kicks in, new sources are needed to fuel your growth even further. But in Cyclone's case, we couldn't use the same bootstrapping tactics that had worked for us the first time around. First, having already blown our savings, we had no other cash reserves to dip into. And the chances of finding programmers and artists who would work—if only for a few months—on new games for free didn't seem too promising. That meant we'd have to look for outside investors who'd ante up the needed cash.

Finding reliable investors can be a long and grueling process, and there are many approaches you can take to round them up. If your company is just getting started and needs to raise cash to fund a quick spurt of growth, a great way to do it is to sell a percentage of the profits from the games your investors are putting money into. In other words, someone gives you money to fund a game and in exchange you give them a percentage of the profits that the game generates. (Cyclone uses a variant of this strategy: We borrow a chunk of cash but agree to pay back the original loan when we've struck a publishing deal for the game that the money is funding. In addition, we give our investor a small percentage of the game's future profits as an interest rate.)

From a developer's perspective, there are a lot of advantages to this kind of arrangement: Offering investors equity pieces in individual games allows them to get a much quicker return on their investments than had they bought equity in your entire company. It's Cyclone's intention to use our investor's money to fund a title for no more than five months—by that time, we think we can find a publisher that will pick up the rest of the game's development and will also let us pay back our original loan a short while later.

The investor's money is at risk for only about six months and he or she in turn gains a piece of profits that can be cashed in about a year and a half down the road, when royalties for the game's sales begin to flow in. On the other hand, if the investor had bought equity in Cyclone as a company, rather than a single game, at what point would he or she recoup the investment? When we go public? When we're bought out by a larger company? If either of these pleasant scenarios ever come to be (and statistically speaking they won't), then they're almost certainly many years away and represent a very long-term return to our investors. And putting money at risk for such a long time can scare potential investors away.



But the biggest reason we try to sell equity in Cyclone's games and not the company itself is that we see our company's equity as being its lifeblood, and we hate the idea of giving away pieces of the company when it's still so young just to get a game or two started. It might be tempting to give someone a significant piece of the pie when they're offering you cold cash, but if you give up equity just getting yourself off the ground, what will be left to offer investors (and for yourself) when your company is more established but still needs more funding to grow even further? That's why we think equity investments should be reserved for strategic cash infusions that allow your company to compete on an entirely new level and not just fund one of what will be many games you end up making.

Recruiting Great People

When you're a small company, and you're not sure how you're going to grow and what kind of staff you're going to need, it's convenient to draw from the large talent pool of freelancers and contractors working in the industry today. For instance, a large part of the character animation and cinematic sequences called for in our first game is being done by a small shop in Portland whose principals formerly worked on TV's California Raisins and Domino's Pizza Noid.

We would never be able to hire these talented people on staff because they would be too expensive as permanent employees, and they like being in business for themselves anyway. Also, while their skill set and interests fit our current project well, they may not be as well suited to other games that we'll be developing down the road. It works to hire freelancers on a project-by-project basis until we know what exact skills will suit the company in the long run.

On the other hand, one of Cyclone's most important goals is to build up our internal staff so we have a broad and strong talent base within the company. We want to do this for a couple of reasons. First, it's generally cheaper to hire employees rather than pay contractors, as long as you know that the employees' skill sets fit the kinds of projects you'll be doing down the road.

Second, we want to bring talented people into Cyclone's permanent team because we think it will make the company more attractive to publishers we work with and possibly to bigger companies that might want to acquire us down the road. Finally, hiring true employees makes the company seem a lot more like a genuine team instead of a collection of hired guns, and that's good for morale. In the long run, it makes sense for us to bring people into the company rather than relying too much on independents.

It's not a secret that hiring great people—smart, creative, driven, and good team players—is the single most important thing that you can do for your company. But the issue that nonetheless faces Cyclone Studios and just about any other aspiring game developer out there is how to attract quality people to your company when you still haven't grown to the point of having the trappings of wealth, power, and prestige.

Certainly, recruiting talent can be a challenge when you're small; recently, for instance, Cyclone recruited a young programmer right out of college. We felt good about his technical skills, saw that he was motivated, and thought his personality would fit in well with the team we were building. But on the day he was supposed to start, he called to tell me he had just interviewed and accepted a job with Crystal Dynamics, one of the hot game developers and publishers these days. Apparently Crystal, which has about $20 million in funding and has been on a hiring spree for months, was able to lure this programmer away with a higher salary and the air of an established company.

In this particular person's case, committing to Cyclone and backing out at the last minute was not exactly the sign of a sterling character, so we weren't too disappointed by the loss. But it still leaves you wondering how a small, struggling developer like Cyclone can compete with the Crystal Dynamics of the world. In some cases, we just can't—there will always be people that follow the highest possible salaries and other perks that only bigger companies can offer.

At the same time, there's an entirely different class of talented, motivated people that small developers stand a good chance of attracting—an even better chance, perhaps, than those bigger companies that seem to have all the money in the world. The video game industry is at its heart an industry of fine artisans—programmers, artists, musicians, and game designers whose greatest satisfaction is their own personal sense of contribution and impact on the games.

In bigger companies, this sense of impact diminishes more and more—work is divided and subdivided again among big development teams, people can be shuffled around from one project to another, games are designed and managed by committees or layers of management, seniority often determines who works on the best projects, politics come into play, and so on and so on. The result is that talented, dedicated people end up feeling like some cog in a wheel instead of an essential and valued part of the team.

While these headaches can be commonplace in bigger organizations, they're hardly issues for small, entrepreneurial shops. At Cyclone, for instance, we've attracted (and hope to continue attracting) strong, quality people by offering them a big degree of responsibility and impact on the work the company does. Everyone here knows that he or she can make a huge contribution to the fun, personality, and ultimate success of each of our games. That's incredibly appealing to the kind of people who are focused on making world-class games and less concerned with their 401K plans. It's in this small company atmosphere that talented people can do some of the best, most inspired work of their careers. Consequently, it's an environment that many prefer to join.

While bigger companies may initially be able to pay their employees higher salaries and assemble long lists of fringe benefits, my bet is that Cyclone and other small, growing game shops can actually offer their people a better financial deal in the long-run than many could expect from more established companies as well. For instance, we give our team members fairly significant royalties on the games they produce, which can be more lucrative than what they'd make as just another programmer or artist at a bigger company.

By emphasizing these two fundamental benefits to potential employees—full impact on their work and the potential for significant profit sharing—we're betting that it's possible for even the smallest development shop to compete with the industry's giants in attracting great people. All you have to do is convince people you're genuinely committed to giving these opportunities once they come on board, and then follow through.

Keep Your Eye on the Ball

There you have it—the official Cyclone Studios guide to starting an independent game development company. Again, though mine is only one of potentially many perspectives, I hope to have given any aspiring game makers a better idea of some of the important challenges and solutions that go into building a small game shop from scratch. If you're in the same position Cyclone is in—either building or thinking about building a company step-by-little-step—I hope you'll find some useful information here. ■

Helmut Kobler joined The 3DO Company in 1992 as the second person in its then-fledgling marketing department. He couldn't resist the temptation of making games for long, though, and left 3DO to form Cyclone Studios in late 1993. He can be reached via e-mail at cyclone@aol.com or through Game Developer *magazine.*

Back

The Mysterious Mode 13h

So, what's so great about Mode 13h? Many game developers like it because its memory configuration makes it easy to manipulate, and it supports 246 simultaneous colors and 262,144 different hues. And, best of all, because it lacks the planar architecture found in the EGA modes, it's fast to boot.

A long time ago, there was a planet called Earth. This planet was inhabited by the first humans. These were curious creatures. They built many machines. They were masters of physics and the arts, and they learned to control all aspects of their environment. In the third epoch of their existence, they took to the stars to colonize the galaxy.

They recorded all their technology on magnetic and optical disks, which we have today. However, much of the information has been lost. One of the greatest losses was the workings of a special graphics mode, supported by a popular computer of the late 20th century called the IBM PC. This mysterious graphics mode was used by thousands of software engineers to develop video games, which were used as a form of entertainment. Game developers used this mode because it allowed them to draw images on the screen at unbelievable rates and with myriad colors.

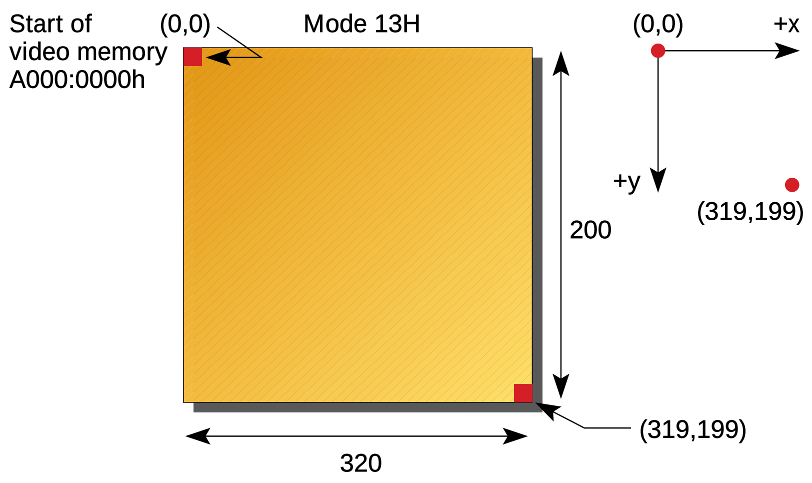
Using our 35th century technology, we have been able to restore the lost data, and now for the first time, we can read about this mysterious graphics mode, 13h. This information is valuable, and great care must be taken by those who wield it. If you choose to read these passages, may all the games you make be very cool.

The Lay of the Land

Mode 13h is the best overall graphics mode that a standard VGA card supports. The mode has a resolution of 320-by-200 and supports 256 colors. Mode 13h is attractive to game programmers for several reasons: its memory configuration makes it easy to manipulate, it supports 256 simultaneous colors, each of which can take on 262,144 different hues, and it's fast because it lacks the planar architecture found in the EGA modes.

Figure 1 shows Mode 13h's 320-by-200 matrix of pixels. Each pixel represents a single dot on the video screen. By turning these pixels on and off, we create images on the video screen. If you are familiar with any of

Figure 1. The Organization of Mode 13h

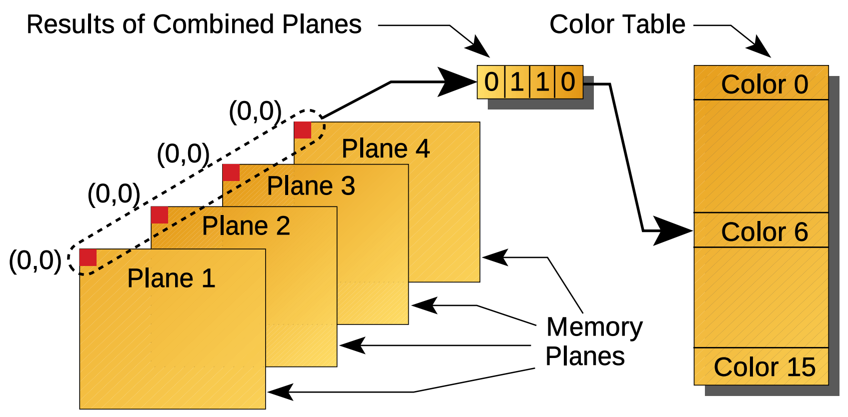


by André LaMothe

Mode 13h is an obscure graphics mode that few game developers have mastered. But those who've uncovered the mystery claim it's the best overall graphics mode around.

the EGA video modes, you will recall that video memory on the EGA is planar. In other words, each pixel is composed of components extracted from separate memory planes, as shown in Figure 2. This is not the case in mode 13h. Mode 13h is one linear region of memory that starts at memory location

Figure 2. The Planar Organization of the EBA Modes



One bit is extracted from each memory plane and combined into a single nibble that is used as an index into the color table.

A000:0000h (segment A000h, offset 0000h) and ends at A000:F9FFh.

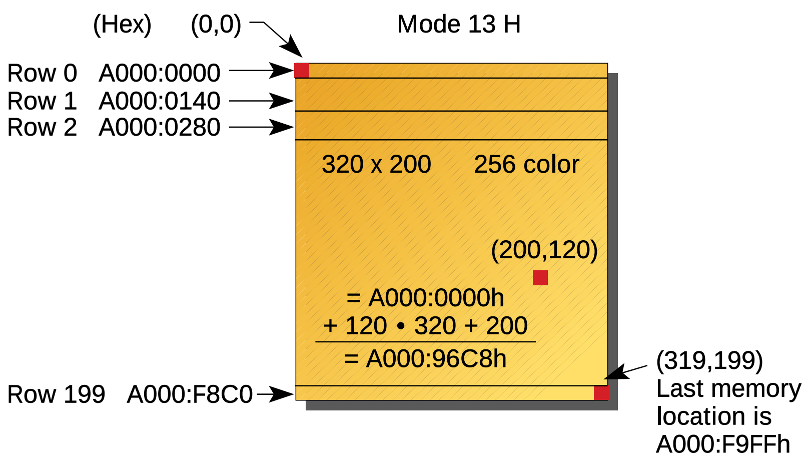
So, mode 13h is exactly 64,000 bytes long. But, what is the relationship between the 64,000 bytes and the 320-by-200 matrix? One byte exists for every pixel on the screen. If one byte exists for every pixel, there should be 320-by-200 (64,000 bytes) making up the screen memory—and, isn't that a coincidence, there is! Take a deep breath, and let that sink in. In essence, all we have to do is point a pointer to video memory at A000:0000h and access video memory as an array. (Finally, something on the PC that makes sense!)

Figure 3 shows the memory layout of mode 13h and some of the row addresses. As you can see, each row of pixels is exactly 320 bytes from the previous row. So, to plot a pixel at any (X,Y) location, you multiply the Y coordinate by 320 and add the X coordinate. Use the final index as an offset from A000:0000h and write your single byte representing the color you want at this point.

If the memory configuration alone isn't enough to make you happy, there's more. Mode 13h supports 256 simultaneous colors on the screen from a palette of 262,144. These colors are implemented by a Color Lookup Table. You can think of this table as a collection of 256 paint buckets that can have any color paint in them you wish. When you write a 26 into screen memory, the color you see on the screen will be the color that is in bucket 26. This is called color indirection, and it is a very powerful method of representing colors within a graphics system. We will cover the Color Lookup Table in more detail in a moment.

Now that you have the overall view, let's see how we can get into mode 13h in the first place. Also, before I forget, I used Microsoft's C/C++ 7.0 for all these examples, but with simple changes, any compiler should work.

Figure 3. The Memory Configuration of Mode 13h



Getting in the Mode

You can get into mode 13h a few different ways. Within Microsoft's graphics library is a function called _setvideomode(). You can use this function to change the video mode to any mode supported by Microsoft. Borland's compiler has a similar function. As an example, let's write a simple program that uses _setvideomode() to get into mode 13h and back out. This program is shown in Listing 1.

If you compile and execute Listing 1 (remember to link in the graphics library), your PC will be in mode 13h. To exit the program, press any key. Using a library function to get into mode 13h is fine, but we need a way that doesn't depend so much on the C compiler's graphics library. A call to the video BIOS will put us into mode 13h. The video BIOS interrupt is INT 10h, and it has many different functions. We are interested in function 0, which is the set_video_mode function. Here are the parameters you need to execute function 0:

```
Sub-Function 0
   AH = 0
   AL = video mode number (13h)
   returned values(none)
```

To use this function, we need a way to call the video BIOS. We can use the

Listing 1. Entering Mode 13h with Microsoft's Graphics Library

```
#include <stdio.h>
#include <graph.h>
#include <conio.h>

int main(void)
{
_setvideomode(_MRES256COLOR);

printf("\nHello world from mode 13h.");

getch();

_setvideomode(_DEFAULTMODE);

} // end main
```

inline assembler, or we can use the C interrupt function `_int86()`. To cover all bases, I will provide code for both. Listing 2 is a complete program that has functions based on the inline assembler as well as the `_int86()` function.

I may be getting a little obsessive by showing you all these ways to get into mode 13h, but I want to make sure you can do it! Notice the defines within Listing 2. You can use these defines to get into mode 13h, to get back out of it, and to get back to the 80-by-25 text mode that DOS is in. These defines are:

```
#define VGA256 0x13
// 320x200x256
#define TEXT_MODE 0x03
// the default text mode
```

Using these defines will make the call to the `Set_Video_Mode()` easier. The `Set_Video_Mode()` function is a direct connection to the video `BIOS`, so you can send any parameter you want! Now that we know how to get into mode 13h (and hopefully back out), let's cover the Color Lookup Table in more depth.

Mixing Colors with the Palette

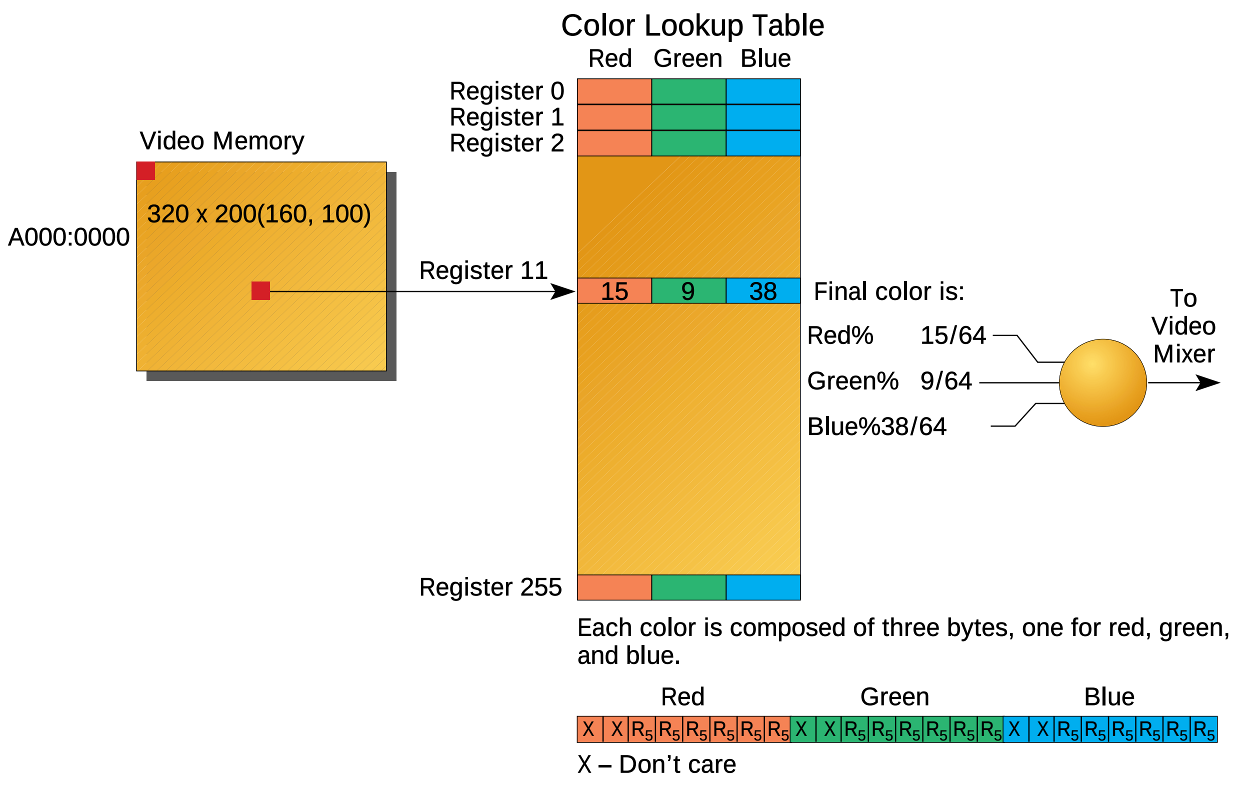
As we discussed previously, the Color Lookup Table comprises 256 color registers. Each holds the RGB (red, green, blue) values that make up each color, as shown in Figure 4. Each RGB component is 8 bits wide, but you use only the first six bits to generate the color for each register. So, each primary color can have 26 or 64 different shades for a total of 218 or 262,144 colors on the standard VGA card.

Here's how the Color Lookup Table works. When you plot a pixel on the screen by writing bytes into the video buffer, the pixel value you write becomes an index into the Color Lookup Table. If you write a 55 into the video buffer, the actual color that you see will be the combined RGB components that exist in color register 55. This opens up an intriguing possibility. By changing the RGB values, the pixels on the screen will change,

Listing 2. Setting the Video Mode to Mode 13h.

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

#define VGA256    0x13       // 320x200x256
#define TEXT_MODE 0x03       // the default text mode

void Set_Video_Mode(int mode)
{

// use the video interrupt 10h to set the video mode to the sent value

union REGS inregs,outregs;

inregs.h.ah = 0;                     // set video mode sub-function
inregs.h.al = (unsigned char)mode;   // video mode to change to

_int86(0x10, &inregs, &outregs);

} // end Set_Video_Mode

void Set_Video_Mode_I(int mode)
{

// use the video interrupt 10h to set the video mode to the sent value

// use the inline assembler

_asm
   {

   mov ah, 0             ; sub - function 0 - set video mode
   mov al, BYTE PTR mode ; move the video mode into al
   int 10h               ; do the interrupt

   } // end asm

} // end Set_Video_Mode_I
/
int main(void)
{
// set video mode to 320x200 256 color mode

Set_Video_Mode(VGA256);

printf("\nHello again!!!");

// wait for keyboard to be hit

while(!kbhit()){}

// go back to text mode

Set_Video_Mode(TEXT_MODE);

} // end main
```

Figure 4. Color Lookup Table Architecture



too. You can change the RGB values by changing the color registers using special ports on the VGA card.

Basically, we need to select the color register we wish to change and alter each RGB component of the color. The ports we need to communicate with on the VGA are:

```
#define PALETTE_MASK        0x3C6
// the bit mask register
#define PALETTE_REGISTER_RD  0x3C7
// set read index at this I/O
#define PALETTE_REGISTER_WR 0x3C8
// set write index at this I/O
#define PALETTE_DATA        0x3C9
// the R/W data is here
```

To change a palette register, we set the palette mask to 255 to ensure all bits in the register selection are valid. Then we write the index of the color register we want to modify to the palette write register (or the palette read register if we wish to read the RGB values). Then the data is written to or read from the data register at 0x3C9. We will see functions that will both read and write to the color registers in a moment, but it might be a good idea to create a little structure to encapsulate the RGB components of a color so we don't have to pass around three pointers, and so on. Here is a color structure I propose to use:

```
// this structure holds a RGB triple
// in three bytes

typedef struct RGB_color_typ
    {

    unsigned char red;
// red  component of color 0-63
    unsigned char green;
// green component of color 0-63
    unsigned char blue;
// blue component of color 0-63

    } RGB_color, *RGB_color_ptr;
```

Having this structure will make life much easier when we write the functions that change the color registers.

Writing to a Color Register

To write to a color register we must know two things: the color register we wish to change and the RGB components of the color we want to change it to. Once we know these things, we can write to the color register using this function:

```
void Set_Palette_Register(int index,
RGB_color_ptr color)
{

// this function sets a single color
// look-up table value indexed by index
// with the value in the color structure

// tell VGA card we are going to
// update a palette register

_outp(PALETTE_MASK,0xff);
// tell vga card which register we
// will be updating
_outp(PALETTE_REGISTER_WR, index);

// now update the RGB triple, note the
// same port is used each time

_outp(PALETTE_DATA,color->red);
_outp(PALETTE_DATA,color->green);
_outp(PALETTE_DATA,color->blue);

} // end Set_Palette_Register
```

To use the function we would define a color structure like this:

```
RGB_color color_1;
```

Then, we would set the fields of the structure and make a call to the function. For example, say we wanted to change color register 134 to a bright red. We might do the following:

```
color_1.red = 255;
color_1.green = 0;
color_1.blue = 0;

Set_Palette_Register(134,(RGB_color_pt
r)&color_1);
```

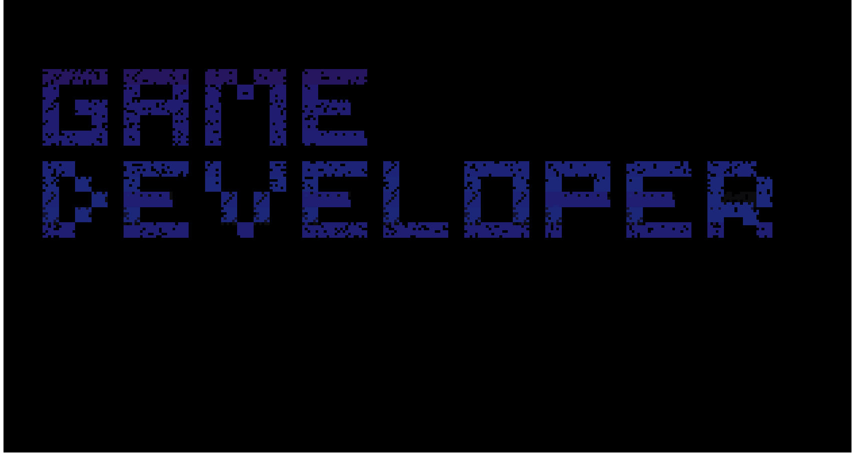
Reading from a Color Register

Reading the RGB components of a color register is just as simple. Instead of writing data to the data port we simply read it back in the order of RGB. The first read of the data register will always be the red component of the color, the second read will always be the green component of the color, and the final read will always be the blue component of the selected color register.

Here's a function that reads a color register and stores the RGB values in the color parameter:

```
void Get_Palette_Register(int index,
RGB_color_ptr color)
{

// this function gets the data out of
// a color lookup register and places it
// into color
```

Figure 5. Demo Program Showing Mode 13h and Color Rotation



```
// set the palette mask register
_outp(PALETTE_MASK,0xff);
// tell vga card which register we
// will be reading

_outp(PALETTE_REGISTER_RD, index);

// now extract the data

color->red  = _inp(PALETTE_DATA);
color->green = _inp(PALETTE_DATA);
color->blue = _inp(PALETTE_DATA);

} // end Get_Palette_Register
```

The interface of `Get_Palette_Color()` is the same as `Set_Palette_Color()`, but the results are different. When you call the `Get_Palette_Color()`, the pointer to the color structure is filled with the RGB values of the sent index. For example, say we wanted to extract the color components of color register 67:

```
Get_Palette_Register(67,
(RGB_color_ptr)&color_1);
```

After the call, `color_1` would have the RGB values of color register 67. Later we will see a program that uses these functions, but now let's learn how to plot pixels on the screen.

Blasting Pixels

As we learned, mode 13h's video memory is one big continuous array of bytes in which each byte represents a single pixel. The bytes are arranged in 320 columns and 200 rows. To plot a pixel at a position (X,Y), we must multiply the Y coordinate by 320 and add the X coordinate. This will give us the final offset from video memory at which to plot the pixel. Before we write a function to plot pixels on the screen, we should create a global variable that points to the video memory so we can access it like an array. We can create this global variable with this definition:

```
unsigned char far *video_buffer = (char
far *)0xA0000000L; // vram byte ptr
```

This statement creates a pointer to the video memory that we can use as a base address for functions. Now, let's write a function that plots a single pixel on the screen.

Plotting Pixels

To plot a pixel we need to know the X and Y location along with the color. We then use this information to create a final address to write the pixel to. Here's the code to do it.

```
void Plot_Pixel(int x,int y,unsigned
char color)
{

// plots the pixel in the desired color
// each row contains 320 bytes, there-
fore multiple Y times the row and add x

video_buffer[y*320+x] = color;
} // end Plot_Pixel
```

Listing 3. A Mode 13H Demo Program (Continued on p. 48)

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <math.h>

#define VGA256              0x13  // 320x200x256
#define TEXT_MODE           0x03  // 80x25 text mode
#define PALETTE_MASK        0x3C6 // the bit mask register
#define PALETTE_REGISTER_RD 0x3C7 // set read index at this I/O
#define PALETTE_REGISTER_WR 0x3C8 // set write index at this I/O
#define PALETTE_DATA        0x3C9 // the R/W data is here
// demo dependent defines
#define BLOCK_SIZE          6     // size of blocks
#define COLOR_BASE          16    // color base of rotation banks
#define GAME_ROWS           11    // number rows in game developer title
#define GAME_COLUMNS        45    // number of columns
#define GAME_X0             20    // origin of where to draw title
#define GAME_Y0             50

// this structure holds a RGB triple in three bytes
typedef struct RGB_color_typ
        {
        unsigned char red;    // red   component of color 0-63
        unsigned char green;  // green component of color 0-63
        unsigned char blue;   // blue  component of color 0-63
        } RGB_color, *RGB_color_ptr;

void Set_Video_Mode(int mode);
void Set_Palette_Register(int index, RGB_color_ptr color);
void Get_Palette_Register(int index, RGB_color_ptr color);
void Plot_Pixel_Fast(int x,int y,unsigned char color);

unsigned char far *video_buffer = (char far *)0xA0000000L; // vram byte ptr

// this is the image that will be displayed, put in anything you want
// the "."´s are for blanks and the "0"´s are for solid
char *game[GAME_ROWS]={

                     "0000.0000.00.00.0000........................",
                     "0....0..0.0.0.0.0...........................",
                     "0.00.0000.0...0.000.........................",
                     "0..0.0..0.0...0.0...........................",
                     "0000.0..0.0...0.0000........................",
                     "............................................",
                     "00...0000.0...0.0000.0....0000.0000.0000.000.",
                     "0.0..0....0...0.0....0....0..0.0..0.0....0..0",
                     "0..0.000...0.0..000..0....0..0.0000.000..000.",
                     "0.0..0.....0.0..0....0....0..0.0....0....0.0.",
                     "00...0000...0...0000.0000.0000.0....0000.0..0",};

void Set_Video_Mode(int mode)
{
// use the video interrupt 10h to set the video mode to the sent value
  union REGS inregs,outregs;inregs.h.ah = 0;                    // set video mode
sub-function
```

The body of the function is literally one line! We would have previously set the video mode to 13h with a call to `Set_Video_Mode()`, so to use the function, we would just call it with the desired parameters. For example, if we wanted to plot a pixel using color register 1 at the location of (100,100), we would make the following call:

```
Plot_Pixel(100,100,1);
```

And presto! A little dot appears on the screen at (100,100).

Reading Pixels

Another important thing to be able to do in a graphics mode is to read from the video buffer. Reading a pixel from the video screen is necessary for collision detection and image-processing algorithms. Reading pixels in mode 13h is a snap—we just do things in reverse. We can almost use the exact same code as the plot function. We just do a return of the data addressed by the coordinates instead of assigning to it.

Here is the function:

```
unsigned char Get_Pixel(int x,int y)
{

// gets the color value of pixel at
// (x,y) from the screen and returns
it

return video_buffer[y*320+x];

} // end Get_Pixel
```

If we wanted to see what pixel value was at location (50,50) we could write the following code.

```
if (Get_Pixel(50,50)==1)
  {
  // do something
  } // end if
```

Faster, Faster, Faster!

Plotting pixels is so important that you must plot them as quickly as possible. So, we must optimize the pixel plotting function as much as we can. Let's see if we can't squeeze some more performance out of it.

When doing graphics on a PC, we must minimize the amount of math we do to process visual elements. If we analyze `Plot_Pixel()`, we see that there isn't much math to optimize—but there is one snippet. We can use shifting to perform the multiplication on this snippet instead of the scalar multiplication. Let me explain.

We need to multiply Y by 320. We can do this in the following way: 320 = 256 + 64. Now, if we multiply Y by 256 and add that to Y * 64, the result will be Y multiplied by 320. Hello??? Why multiply Y by two numbers? Because we can use binary shifting to accomplish the multiplication. Remember, shifting to the left in binary is like multiplying by 2, and we can use this knowledge to write a fast `Plot_Pixel()` function.

A Faster Pixel Plotter

Because we can achieve multiplication by binary shifting along with addition, we will rewrite the pixel plotter to multiply Y by 320. Here is the code:

```
void Plot_Pixel_Fast(int x, int
y,unsigned char color)
{

// plots the pixel in the desired color
a little quicker using binary shifting
// to accomplish the multiplications

// use the fact that 320*y = 256*y +
// 64*y = y<<8 + y<<6

video_buffer[((y<<8) + (y<<6)) + x] =
color;

} // end Plot_Pixel_Fast
```

This function works the same as the previous function except it is about 50% faster, if not more (this will depend on your machine). We have covered all the main points of the mysteries of mode 13h. Now let's put

Listing 3. (Continued on p. 49)

```
inregs.h.al = (unsigned char)mode;  // video mode to change to
_int86(0x10, &inregs, &outregs);
} // end Set_Video_Mode
/void Set_Palette_Register(int index, RGB_color_ptr color)
{
// this function sets a single color look up table value indexed by index
// with the value in the color structure
// tell VGA card we are going to update a palette register

_outp(PALETTE_MASK,0xff);
// tell vga card which register we will be updating
_outp(PALETTE_REGISTER_WR, index);

// now update the RGB triple, note the same port is used each time
_outp(PALETTE_DATA,color->red);
_outp(PALETTE_DATA,color->green);
_outp(PALETTE_DATA,color->blue);
} // end Set_Palette_Register
void Get_Palette_Register(int index, RGB_color_ptr color)
{

// this function gets the data out of a color lookup register and places it
// into color
// set the palette mask register
_outp(PALETTE_MASK,0xff);
// tell vga card which register we will be reading
_outp(PALETTE_REGISTER_RD, index);

// now extract the data
color->red   = _inp(PALETTE_DATA);
color->green = _inp(PALETTE_DATA);
color->blue  = _inp(PALETTE_DATA);
} // end Get_Palette_Register
void Plot_Pixel_Fast(int x,int y,unsigned char color)
{
// plots the pixel in the desired color a little quicker using binary shifting
// to accomplish the multiplications
// use the fact that 320*y = 256*y + 64*y = y<<8 + y<<6
video_buffer[((y<<8) + (y<<6)) + x] = color;
} // end Plot_Pixel_Fast

void Draw_Game(void)
{
// this function randomly fills up the game developer title with 11
// different shades of blue that are then color rotated by means using
// the color look up table
int x,y,index,clock=0;
RGB_color color,save_color;
// create blue color palette

for (index=COLOR_BASE; index<COLOR_BASE+GAME_ROWS; index++)
    {

    color.red   = 0;
```

everything together into a demo that does something!

Demoland in Mode 13h

This demo program plots a bunch of random pixels in a single color. Using the color register functions, it then changes the color of all the pixels on the screen. The final demo is a little more exciting. It draws the words "Game Developer" on the screen and then uses a technique called "color rotation" to make the colors look like they are moving. Figure 5 illustrates the output of the demo. The demo's source code can be found in Listing 3.

There isn't much compiler-dependent code in the demo. You may have trouble using the `_int86()` and `kbhit()` functions, but all compilers probably have equivalent functions with almost the same names.

The ancient Earth technology of the mysterious mode 13h has been uncovered for all to see. Few beings have seen what your eyes have seen—the configuration of memory, the Color Lookup Table, and how to plot pixels at unimaginable speeds. Use these powers wisely and teach all that cross you path so that they may share in the knowledge—and create cool games. ■

Andre LaMothe holds degrees in math, computer science, and electrical engineering, and worked in neural networks, three-dimensional graphics, virtual reality, and robotics before becoming a game developer. He is the author of the book Tricks of the PC Game Programming Gurus *(SAMS Publishing, 1994). He's now writing a second book for beginning game programmers. You can contact him through* Game Developer *magazine.*

Listing 3. (Continued from p. 48)

```
    color.green = 0;
    color.blue  = (index - COLOR_BASE + 1)*5;

    Set_Palette_Register(index,(RGB_color_ptr)&color);(



    } // end for color

// do this until user hits a key

while(!kbhit())
     {
     // plot a pixel somewhere in the game developer title
     x = rand()%GAME_COLUMNS;
     y = rand()%GAME_ROWS;
     // test if there is a block there
     if (game[y][x] == ´0´)
        {
        Plot_Pixel_Fast(x*BLOCK_SIZE+GAME_X0+rand()%BLOCK_SIZE,
                        y*BLOCK_SIZE+GAME_Y0+rand()%BLOCK_SIZE,
                        y+COLOR_BASE);
        } // end if

     // rotate the colors
     // this is an effect where by we shift the values of one color
     // register into another, this results in a "bucket brigade"
     // effect that makes the colors look like they are moving
     if (++clock==200) // wait 200 cycles before each rotation
        {
        // save the first register in sequence
        Get_Palette_Register(COLOR_BASE,(RGB_color_ptr)&save_color);
        // rotate the colors
        for (index=COLOR_BASE+1; index<COLOR_BASE+GAME_ROWS; index++)
            {
            // place the nth color register in the (n-1)th
            Get_Palette_Register(index,(RGB_color_ptr)&color);
            Set_Palette_Register(index-1,(RGB_color_ptr)&color);
            } // end for index
       // complete the circle
       Set_Palette_Register(index-1,(RGB_color_ptr)&save_color);
       // reset counter clock
       clock=0;
       } // end if clock
     } // end while
} // end Draw_Game

int main(void)
{// set video mode to 320x200x256
 Set_Video_Mode(VGA256);
Draw_Game();
// reset video to text mode
Set_Video_Mode(TEXT_MODE);
return(1);
} // end main
```

Back

Breaching the Rebel Base

by Wayne Sikes

Slice and dice LucasArts' Rebel Assault to meet your needs. Find out how—and more—when we place it on the Chopping Block.

The test of time has proven well for Rebel Assault—its overall quality still shines.

Since this is the first appearance of the Chopping Block, please indulge me while I describe some things you will see in this column. But first, let's get something out of the way—what you will *not* see in this column.

If you want general game reviews, a description of Sim's weapon modes, the types of missiles you can fire, the sound cards supported, or how well a game's documentation and manuals are written, don't look here.

If you want an overview of a game's internals (executables and data files), a general commentary on how well the graphics and sound are implemented, a description of the game engine (and whether or not it is based on engines in previously released products from the same manufacturer), or how well the user interface is implemented, do look here. I will also try to describe how you can tailor a game to your liking, that is, make flight modes easier, increase the damage your ship can sustain, give you more and better weapons, and so on.

Depending on how the game is written, this is not always easy, and sometimes it can't be done at all. (Some game companies encrypt their data and executables using data compression routines and other methods to make the internals use less disk space; it also locks out most hackers.) I'll try to give you all the information I can while at the same time not violate any copyright laws. I'll also assume that not every reader has many years of C++ and assembly language programming experience under his or her belt.

When reviewing a game for this column, my first step is to look at the overall quality of the graphics, sound, and user interface. From there, I analyze the game's internals. I don't look at every byte of code because you can miss the big picture using that approach, and it takes too much time.

I look at how well the internals are written, the overall structure of the game engine, how well the graphics and sound are used, how well the user interface works, and how usable the game really is. The last evaluation point is totally subjective. I feel that most games should contain the controls for making them playable by anyone from my two-year-old daughter to a Ph.D. computer engineer.

Let's Get On With It!

The first game on the chopping block is Rebel Assault by LucasArts Entertainment Co. This game was in the making for several years, and it shows. The overall quality of Rebel Assault shines, with few detractions.

On first perusal, I was very impressed with the three-dimensional object rendering, and the sound was excellent. My amazement turned to irritation when I discovered I couldn't get through the fourth training mission (the Planet Kolaador sequence), and there was no way to save my training up to that point. Even worse, my joystick had only rudimentary control sensitivity adjustments, making me a worse pilot than I already am!

The primary Rebel Assault executable is ASSAULT.EXE. This file is about 212,000 bytes in length and contains the game engine. A game engine is the executable code that controls all primary game activities, such as providing animation for the graphics display, monitoring user interface devices (keyboard, mouse, and joystick), activating and routing information to the sound devices (your PC speaker or a sound card), and providing artificial intelligence for game play.

Memory management is provided by Tenberry's DOS4GW.EXE. Programs such as Rebel Assault often require a high memory overhead because graphic data must be continuously streamed from the CD-ROM and stored in upper memory (memory above the conventional 640K of RAM). This streaming effect allows for higher display frame rates and seamless transitions between scenes. The DOS4GW.EXE memory manager created problems for LucasArts because it refused to run correctly on some computers. The result of these problems was that Rebel Assault would crash. LucasArts has issued revisions of DOS4GW.EXE in some of the game patches.

A general analysis of ASSAULT.EXE reveals a cleanly written, well organized, and well thought out game engine. The modularity of the game shows the designers obvious intent to port it to various platforms after release of the IBM version (perhaps Sega CD, Macintosh, and 3DO). The game was so cleanly written that analysis of the internals and location of the primary control variables was about the easiest analysis job I've done.

By the way, that's intended as a compliment to the LucasArts team. Reviewing source code written by someone else is tough enough and reviewing executable code is much tougher. Any group who can write and compile executable code like this deserves some praise. Despite the memory management problems previously mentioned, the ASSAULT.EXE code appears to be stable enough to run under other operating environments, such as OS/2.

Rebel Assault was written in C using the Watcom C 386 run-time development system. C and C++ are the languages of choice for writing games in today's market. (I've seen a few commercial games written in BASIC and Pascal, but not many.) One advantage of C is that you can easily write modular, efficient code. This modularity extends to the data structures used in the game.

Rebel Assault uses a single data structure for controlling the primary mission variables such as joystick sensitivities, targeting accuracy, damage accumulation, and point scoring increments. Listing 1 shows an example of this data structure. (This C `struct` was sourced from a utility I wrote called RAEASY, not the ASSAULT.EXE source code.) Even if you are not familiar with any type of programming language, take a look at this code to see the various things you can alter. Most variable names explain their function.

The first five bytes in the structure (`mission_name` variable) give the internal name of the mission in ASCII text (null-terminated `char` string for you C programmers). Twenty-one in all, some missions have A and B parts. The missions are ordered as: 1A, 1B, 2, 3, 4A, 4B, 5A, 5B, 6, 7, 8, 9A, 9B, 10, 11, 12, 13, 14A, 14B, 15A, and 15B.

The four joystick sensitivities are grouped together. The `roll_sens` and `lift_sens` variables appear to control the rotational properties of your vehicle,

Listing 1. RAEASY C Language Data Structure

```
/*  RAEASY MISSION RECORD STRUCT                                          */
/*  Setup missions[][] as a 2D struct for accessing Rebel Assault mission */
/*  record data. The first array element, missions[x][], accesses the     */
/*  missions according to difficulty.  There are 3 groups of missions -   */
/*  EASY, NORMAL, and HARD.  The second array element, missions[][x],     */
/*  accesses the individual missions under the specified difficulty level. */
/*  Example:  missions[1][2] would access the Asteroid Field Training     */
/*            data structure at the NORMAL level of play.                 */

#define BYTE          unsigned char     /* 8-bit UNSIGNED value */
#define S_WORD        short             /* 16-bit SIGNED value  */
#define EASY_LEVEL       0
#define NORMAL_LEVEL     1
#define HARD_LEVEL       2
#define NUM_DIFFICULTY_LEVELS   3       /* total number of game levels.  */
#define NUM_BYTES_PER_MISSION   31      /* bytes per mission struct */
#define NUM_MISSIONS     21             /* total number of missions */
                                        /* per level.          */

struct     MISSIONS
   {
   char        mission_name[5];   /* Offset 0-4, Mission Name */
   S_WORD      roll_sens;         /* 5-6, Joy=>Ship X-Axis ROLL sens*/
   S_WORD      lift_sens;         /* 7-8, Joy=>Ship Y-Axis LIFT sens*/
   S_WORD      slide_sens;        /* 9-10,Joy=>Ship X-Axis SLIDE sens*/
   S_WORD      drift_sens;        /* 11-12,Joy=>Ship Y-Axis DRIFT sens*/
   S_WORD      targeting;         /* 13-14, Auto Targeting Level */
   S_WORD      missile_damage;    /* 15-16, Missile Damage Increment */
   S_WORD      collision_damage;  /* 17-18, Collision Damage Increment */
   S_WORD      shot_damage;       /* 19-20, Gun Fire Damage Increment */
   S_WORD      kill_points;       /* 21-22, KILL Points */
   S_WORD      time_points;       /* 23-24, TIME Points */
   S_WORD      level_points;      /* 25-26, LEVEL Points */
   S_WORD      bonus_points;      /* 27-28, BONUS Points */
   BYTE        flags0;            /* 29, FLAGS0  */
   BYTE        flags1;            /* 30, FLAGS1  */
   } missions[NUM_DIFFICULTY_LEVELS][NUM_MISSIONS];
```

while slide_sens and drift_sens possibly control the translational (forward and backward) properties. Since these four variables appear to work in tandem, the specific function of each variable isn't easy to ascertain, so just experiment.

You can be damaged by colliding with other objects (canyon walls), enemy missiles, and enemy gunfire. The damage increments are stored in the collision_damage, missile_damage, and shot_damage variables. Damage monitoring is done by reading the damage increment variables each time you sustain damage. The increments are added to internal damage accumulators. When the accumulators reach their maximum damage amount, your character dies.

Targeting difficulty is monitored in the targeting variable. This variable controls how hard or easy it is to target an enemy. You can vary the targeting difficulty from very easy (essentially an auto-targeting mode) to very hard.

All point scoring increments are stored in the primary mission data structure. You get points for killing an enemy, surviving for finite periods of time, and completing a level. You can also get bonus points for superior performance.

The last two variables in the structure (flags0 and flags1) are bit flags. Bit flags are individual data bits that control game functions, such as debugging and vehicle displays, including weapon fire, graphics, weapon type, and so on. Be careful when experimenting with these bit flags; you can easily destroy a mission or make it totally unplayable.

As a final note on the primary mission structure, I found data in ASSAULT.EXE that indicates you can get a debugging and developmental data display of this structure while the game is running. Unfortunately, I haven't figured out how to get the display on the screen yet.

The graphic engine and its associated graphics files worked well. I experienced very few time delays on my 486DX2/50 during periods of heavy graphic rendering. The frame rate was good enough to avoid choppy graphic scenery transitions.

The graphics files are very modular. The graphics consist of two basic file types: animation and checksum. The animation files have a ANM or NUT file suffix. The NUT files are compressed graphic images. The first four file bytes of ANM and NUT are ANIM with an AHDR (animation) or NAHDR (nut type) header description immediately following.

The animation files contain animation frames FRME with each frame consisting of one or more frame objects FOBJ. I wasn't able to discern the exact type of graphics animation system (FLI, FLC, and so on) used by the game. Checksum files have a CHK suffix, and they were probably included to ensure the correct transfer of animation file data from the CD-ROM to RAM. Each animation file has a corresponding checksum file.

Even though the *Star Wars'* scene clips could use some adjustments, the three-dimensional animation graphics in Rebel Assault are excellent.

I've been told that you can change the order of the missions just by rearranging the mission file names stored in ASSAULT.EXE. The game is modular enough that this may indeed be possible. The problem with this technique is that, unless you have the source code, many internal variables you are unaware of will probably be corrupted. If you are serious about your scores, number of kills, and so on, don't try reordering game missions in this manner. Instead, check out the RACHEA.TXT file by Andy Naef (100270.2426@compuserve.com). This file shows you how to access the Rebel Assault cheat mode. Use this mode to skip missions.

Even though the three-dimensional animation graphics were excellent, I can't say the same for the video clips that were interlaced throughout the game. LucasArts basically took scene clips from the *Star Wars* movies and inserted them into the game. The video playback could use some significant work. The color palette was bad and the playback seemed a little jerky.

The sound worked well with my ProAudio Spectrum card, but apparently not as well on some other machines. I talked to many gamers who had problems such as no sound, pops or clicking noises in place of digitized sound, or all the sounds appeared as echoes of the real sounds. Rerunning the REBEL.EXE executable helped some, but I know of several people who still had sound problems even after manually editing the REB.BAT file and inserting data for their particular sound card.

Battling the Asteroid Field

This section can be broken down into two areas. The first area lists things that I would like to see LucasArts add into Rebel Assault. The second gives some things that you can do to alter the performance of the game.

The main problem I found with Rebel Assault was the way the joystick interface worked. At present, joystick control is strictly tied to the values stored in the game's primary mission data structure. LucasArts uses these values to determine part of a mission's difficulty. In the future, the joystick interface should consist of two sets of joystick sensitivity variables. The first set should contain current mission structure variables, and the second should contain a user input set of scale factors. The scale factors would be dependent on the user's joystick and how sensitive he or she would want the vehicle to be to joystick movement. A simpler solution would be to force all joystick sensitivities to user input levels. Both methods would accomplish the same thing—better control of the vehicle. The best joystick interface I've seen was in Velocity's Jet-Fighter II. Why can't Rebel Assault work the same way?

Improve the Game Yourself

Rebel Assault was cleanly written and isn't difficult to modify. You can use either a prewritten editor, such as the one I wrote called RAEASY, to modify the ASSAULT.EXE code, or you can modify it yourself using any general hex editor. Since Rebel Assault is sold only on CD-ROM, any modifications to the game assume you have copied the ASSAULT.EXE file from your CD-ROM to your \rebel subdirectory on your hard drive. (Make sure the read-only bit on the hard drive copy of ASSAULT.EXE is turned off. Use the DOS `attrib` command for this function.)

You may need to modify the REB.BAT batch file to force the game to run your hard drive copy of ASSAULT.EXE while still reading all the game data from the CD-ROM. The following code fragment gives a general REB.BAT batch file set up, assuming your hard drive is C: and your CD-ROM is D:

```
D:
cd \
C:\rebel\assault.exe [command line args]
```

Before discussing code modifications, any alterations you make to the Rebel Assault executable can cause potentially disastrous results. Always backup the ASSAULT.EXE file before you modify it!

I just received a mail message from someone who was lamenting the loss of his modified ASSAULT.EXE file. He worked for a couple of months tweaking the file until Rebel Assault played exactly as he wanted. He had just installed the version 1.7 patch to the game. The patch installation overwrote his original, “perfected” copy of ASSAULT.EXE. He decided he didn’t like the patch and wanted his old ASSAULT.EXE file back.

I previously discussed the game’s primary mission data structure in Listing 1. This data structure can be found in version 1.0 of ASSAULT.EXE at file offset 192278 (hex 2ef16), version 1.4 has this data at file offset 190226 (hex 2e712), and version 1.7 has the data at file offset 183968 (hex 2cea0). To use this structure for modifying the ASSAULT.EXE file, copy the data from Listing 1 into your C program and read the ASSAULT.EXE primary mission data structure into the `missions[][]` structure, starting at the correct file offset. There are several ways you can do this. Here’s an example from RAEASY.C:

```
fseek( FilePtr, (long)FileOffset,
   SEEK_SET );
if(fread(&missions[0][0],sizeof
   (missions), 1, FilePtr) != 1 ) {
      /* Error if gets here /*
      }
```

When you are through altering the `missions[][]` structure, you can save it back to ASSAULT.EXE with an `fwrite()` call.

In general, setting the joystick sensitivities higher (`roll_sens`, `lift_sens`, `slide_sens`, and `drift_sens`) results in greater control. Lowering the value of the damage increment variables (`missile_damage`, `shot_damage`, and `collision_damage`) results in little or no vehicle damage because the internal damage accumulators never get filled. Increasing the `targeting` variable value to a large number such as 10,000 gives you an auto-targeting mode. Setting all joystick sensitivities to zero, all damage increments to zero, and targeting to 32,767 gives you a great Rebel Assault demo. Just pull the trigger, and everything else is done for you!

Now lets tweak the graphics engine for fun. Using a standard text editor, open the REB.BAT file. Look for the line where the ASSAULT.EXE file is called. On the same line following the ASSAULT.EXE name are several command line switches (`/f`, `/t`, `/u`, and so on). Find the `/f` and `/t` switches. Each switch has a number value immediately following it. The `/f` switch controls the graphics frame rate (frames per second). The REBEL.EXE routine sets a maximum value of 15 for this switch. Try increasing this value to 30 or 40. The `/t` switch sets the timing of an internal multitasking clock. This clock controls how often your peripherals (joystick or mouse) are read. Increase this value to around 500. The result of these modifications is a version of Rebel Assault that runs in hyper mode. The graphics will significantly speed up so that you’ll definitely need some help to survive in this mode. You will need a computer that’s capable of running Rebel Assault in a faster-than-normal mode. Gamers having a 486/33 or faster system should have no trouble.

Time to Warp Outta Here

Rebel Assault gives you the best of both worlds. It’s a great action game with fantastic graphics and sound, plus it was so well written that you can easily tweak it to tailor the game play to your skill level. I hope LucasArts continues to provide us with games of this caliber. ■

Wayne Sikes has been a computer hardware and software engineer for the last 10 years. He has an extensive background 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 while working on classified government projects. He has authored numerous computer gaming help utilities. He can be reached via CompuServe at 70733,1562, or Internet at waynesikes@sandia.gov, or through Game Developer *magazine.*

THE ALLIANCE

Rebel Assault

LucasArts Entertainment Co.
P.O. Box 10307
San Rafael, Ca 94912-0307
(415) 721-3394
(415) 456-4381

Rebel Assault is made by LucasArt s Entertainment Co. and usually sells for $63.95. The game requires an MPC level 1 or better computer.

Back

The Once and Future King

by Alexander Antoniades

How Richard Garriot became Lord British, made a lot of money, and went from doing it all to running it all on the same on-going project. Welcome to the world of Origin and its highly successful Ultima series.

When I was told by someone at Origin that Richard Garriot was resigning as head of the Ultima series to become director of product development, I thought he was kidding. He wasn't. To understand the impact of this statement, you really have to know who Richard Garriot—and Ultima—is.

Ultima is probably the best known series on just about any personal computer system. The series originated with Ultima I, which appeared on the Apple II in 1982. Garriot, or Lord British as he's known, is the author who created this series from scratch. When the PC game industry was in its infancy, he was one of the industry's few stars.

He's brought a personal touch to all the Ultima games over the years. The regular cast of characters that made their appearance in almost every Ultima episode was a representation of aspects of Garriot's personality or characters that were closely based on his friends.

Other personal touches include taking inspiration from different real-world events and translating them into the game. For example, the storyline in Ultima V was inspired by the saga of the Dead Sea scrolls. Another example is the Guardian religion in Ultima VII, which is loosely based on the premise of Scientology.

Is this game industry icon losing control of his prize project? Not exactly. From the very beginning, Garriot has been recognized as being the sole author of the Ultima series. While this was true in the beginning, his role evolved as his games did, and his authorship changed.

The Pain Went Away

Of all the games written by Garriot only Akalabeth, his first Apple II game, was written from the ground up. Even Ultima I contained one assembly language routine that was written by a friend of his, although he still wrote, drew, scripted, and scored the entire game. Ultima II and III were also solo endeavors although the same friend now wrote the music as well.

By the time the series reached Ultima IV, the sheer quantity of code was overwhelming, and Garriot needed help. Specialists were required for various aspects of the game, such as the path-finding utility, which allowed nonplayer characters to find their way around the world. Garriot still retained plotting and scripting chores for Ultima IV and V, but with Ultima V, he relinquished the chores of being an artist to professionals. By the time he got to Ultima VII, he needed help with the scripting and world creation.

Since Ultima I through III were almost entirely solo efforts Garriot had learned to work with total control. With Ultima IV, he had to adjust to working with another programmer—this was the first time he started realizing what he was giving up. Programming was the area of creating the game he loved the most, and it was one of the toughest parts to relinquish.

However, as he was handing off work, he began to realize the benefits of working with other people. For example, in Ultima V, he commissioned art from real artists, and the improvements were instantaneous. Garriot never considered himself a writer or artist, he simply did these tasks because there was no one else to do them. Once he saw what professional artists could do for the look of the game, turning over aspects of the creation to specialists became less of a problem.

"The transition of doing it all yourself to doing it as a team was very painful," admits Garriot. "However, once you had a team in place, and especially once you were no longer sharing the duties of both doing it and managing it, the pain went away."

The Interactive Move Studio System

Roles were evolving in the Ultima creation process. The producer is the business manager and determines the overall direction of a project. The director is the daily task master, guiding the team on a direct level. Underneath him are the technical director, who is responsible for code and manages the programmers, and the creative director, who is responsible for the script and the general flavor of the game and story details. The art director manages the art staff and the look of a game as a whole. Last, there is a production assistant, who fills various roles throughout the production process.

Garriot's role until Ultima VII was that of producer, director, and creative director. Now for Ultima VIII, he serves only as producer. Even though he plotted Ultima VII through IX, he has become less and less involved in the actual scripting. For Ultima IX, he will serve as executive producer, similar to the role Gene Roddenberry played in the creation of the *Star Trek* series.

The structure is similar to the movie industry with one major difference. The creative team, which would be the equivalent of the screenwriter, retains control of the scripting. The needs of writing the game are such that the writing is done continuously until the end of the project.

Sandbox Games

Garriot feels that part of the appeal of the Ultima series is that each segment has been a significant improvement over the last. This was a habit he got into early when he wrote Ultima I. When he finished Ultima I, he thought it would be much better if he knew assembly language. When he finished writing Ultima II in assembly language, he knew he could do much better now that he actually knew the language. Early on, he got into the habit of completely starting from scratch, and it has helped the longevity of the series.

With this idea in mind, the design process starts from scratch, which determines the general goals of the project. The base level machine (that is, a 486/33 with 8MB of RAM) is decided so that performance goals can be set. Another part of the process is to make certain basic assumptions about game play. For example, if travel on steeds (such as horses) should be possible, the map must be designed in a certain way from the beginning. Interface changes are also made at this stage, and other additions, such as weather are also added.

A majority of the creation process is defining of the world physics (what will be possible with the way the game code is written). This stage has moved from two thirds of the total creation time in earlier Ultimas to one half the total time currently. Details aren't incorporated into the storyline until the implementation is determined. Garriot refers to this style of game creation as building "sandbox games." Some of the story has to be left open until "you know what toys you have to play with."

An example of this is the harpsichord in Ultima V. In the versions before Ultima V, there were only enough pieces of art to make the necessary objects in the world. In Ultima V, there were enough tiles to actually craft other objects. So Garriot added a harpsichord object to the world. Once the harpsichord was there, it was simply a matter of adding a few lines of code to make the harpsichord sing. Once the harpsichord can play music, just add in some more code so when a special combination of notes is played, a wall will open up and give you a special item that is important for the game.

The creative process is tackled this way, making it easier to determine at the actual time of creation how hard or easy it will be to implement a certain idea. "The details are done in concert with the generation of the physics."

The simplicity of the early Ultimas made adding new additions easier at

Ultima VIII, the last hard-disk based game in the series, broke many of the rules of traditional game programming. Besides leaving out the standard cast of characters, the addition of arcade style sequences didn't go over well with many critics and players. While maintaining that they have no regrets, Origin designers are heading back to basics with the next release.

almost any stage of the design process. The city of Dawn from Ultima II was a city that would appear when the two moons were in alignment with each other. This required a minimal code adjustment to the game. All that had to be done was change the land tile to a city tile in the event that the two moons in the game register a certain value. In Ultima IX, the same effect would require almost a total rewrite of the world map.

This level of complexity requires that there be much more preplanning from the beginning. The movie-making analogy continues as the notion of preplanning becomes more important in the actual creative process of making the game. The creativity goals have become harder to achieve as more preplanning is required.

The Falling Apple

Growing from the original Apple II market, by the time Ultima V came out, the Ultima series loomed over most PC markets. The biggest market sections were divided almost equally between the Commodore Amiga, The Apple II, and IBM PC. Garriot was a tremendous Apple fan and thought Apple would win the PC wars, so he kept the primary development for the Ultima series on the Apple II platform.

This mistake almost cost him the company. Halfway through Ultima VI, it occurred to him that by the time he finished there would be no Apple market to sell the game to. At this point, Ultima VI was almost complete for the Apple IIgs. He immediately stopped development on the Apple and hired some experienced PC programmers to port over the unfinished game.

Overnight, Garriot went from being an old-time Apple hacker to a PC programming novice under pressure to get the game out. Since the move to the PC, he has not actively programmed in any of the Ultima series. Garriot still does not program in C++, the primary programming language used in Ultima series, but due to his hacker background, he can read and understand all the technical aspects of the code well

THE ORIGIN OF ORIGIN

Richard Garriot started programming games in high school, writing Dungeons and Dragons games on a 10 character-per-second teletype machine. Every time the character moved the teletype would print out a 10-by-10 segment that would represent a room with asterisks for walls.

Garriot's first real game, Akalabeth, was the result of programming in BASIC on an Apple II while he worked nights at a computer store in Nassau Bay, where he grew up. The store owner saw the game and told Richard he should publish it. The computer game industry being what it was in those days, the purchase of some ziplock bags and some xeroxed type-written manuals gave him state-of-the-art packaging. He sold the bags in the store for $19.95.

One of those packages made it to California and fell into the hands of California Pacific, a software distributor. It in turn sold the game for $34.95 adding a bigger ziplock bag and a full color manual. Making a $5 royalty off every bag, Garriot ended up making $150,000 for some programming that he did to kill time for three months while he was working at a computer store. Clearly he was on the right track.

With this in mind, Garriot created the first Ultima. Unfortunately, California Pacific was going through financial difficulties at the time and stopped paying royalties for Ultima I. Eventually, California Pacific went out of business, which put him in the market for a new distributor. He found On-Line Systems (which later became Sierra On-Line). On-Line published Ultima II, but Garriot looked for another distributor during contract negotiations.

At that time, his older brother Robert was finishing his masters degree in business and wanted to start a company. Garriot joined his brother and a friend called Chuck Beuche and formed Origin Systems in the Garriots' parents' garage.

Since Richard and Chuck didn't have any obligations in Texas, they agreed to move up to Massachusetts for three years. After that time, all parties agreed they would move to a new location.

But things didn't go according to plan. Origin was growing and more staff members were being added. It soon became clear that the idea of leaving Massachusetts wasn't going to work. All the new staff members were native North Easterners, who probably didn't share Origin's founder's sense of wanderlust. Fearing that Origin would never move to someplace warmer than New England, first Chuck and then Richard moved back to Texas leaving Origin up in Massachusetts.

Chuck stopped working for Origin, but Richard continued. His plan was to continue working on Ultima in sunny Austin and leave the rest of the company up in Massachusetts. But once again, things didn't go according to plan.

According to Richard, it's the location. Nestled between the large companies on the west coast and the large companies on the east coast, Origin used to be the only game company of note in the midwest. Game programmers for a thousand miles in all directions converged on Origin looking for jobs. With this kind of influx of talent, the development team in Austin grew. Eventually, since all the development was done in Austin, all the departments that worked with the development staff eventually trickled down to Austin, until Robert Garriot was the sole Origin employee living on the east coast.

In 1993, Origin was sold to Electronic Arts, which brought the feud between Richard Garriot and Trip Hawkins full circle. Origin had stopped using Electronic Arts as a distributor years earlier, mainly over personal disagreements between Trip Hawkins, Electronic Arts' president, and Richard. Richard responded by parodying Trip with the evil Ultima character, Pirt Snikwah, and made the evil blackrock in the shape of the Electronic Arts logo.

enough to participate in any discussions regarding it.

Intellectual Virtual Reality

Garriot is still very much in the loop of the Ultima series. Even though his role as executive producer doesn't require him to do as much, he's still much closer to this project than others that fall under his jurisdiction at Origin.

The story content has been one of the central areas of concentration since Ultima IV, and the trend will continue. Whereas Ultima I through III were the basic hack and slash dungeon and dragons type game, story development became one of the central focuses of Ultima IV and helped catapult it to be the best-selling Ultima in the series. The current storyline has been planned as far as Ultima IX, which is currently under development—after that, anything can happen.

The mixed reviews that have plagued Ultima VIII don't concern Garriot. When changes are made to the Ultima series, he observed, there are often complaints. A case in point was the change from the numerous keyboard commands in pervious versions of Ultima to a mouse-only interface in Ultima VI.

As for the future, Garriot thinks of the Ultima series as "intellectual virtual reality" and hopes to bring the Ultima series into true virtual reality when the hardware is ready. Ultima VIII and IX are close to reaching the cinematic goal he envisioned when he started developing the series. In the future, all Ultima versions will be CD-ROM based.

In the end, the goals that Garriot established for himself remain the same. He feels he has been driven, not by the necessity to build his own game, but to build the best game. Whether he does it by himself or with the help of a hundred people, that is exactly what he intends to do. ■

Alexander Antoniades is the associate editor of Game Developer *magazine and assistant editor of* OS/2 Magazine. *He can be reached via e-mail at sander@mfi.com or through* Game Developer *magazine.*

Back

That's Life

by David Sieks

Computer graphics bring more to your game than just a cool logo, great sound effects, bright colors, and superheros with bulging muscles, they bring your game to life with a personality of its own.

In the boffo gross-out special effects thriller *The Fly*, Jeff Goldblum's scientist, Seth Brundle, fails—quite messily—to teleport a living creature from one end of the room to the other. The stumbling block is that elusive element that separates a living, breathing lab animal from a gory jumble of offal: life principle. Soul. Anima. The flesh, he discovers, is not life.

The visual artist attempting a representational image or animation is faced with a similar dilemma; there's no easy equation to imbue artwork with life either, though admittedly our failures are easier to clean up than Seth's mistake. The easiest clean-up of all goes to the artist employing computer graphics (oh, if only everything in life came with an Undo feature). The trade-off is that despite the enviable bag of tricks provided by rendering software—or perhaps even because of it—the computer artist might just be more prone than others to stumble over this particular hurdle.

After dutifully drawing the right number of appendages and orifices, we are able to wrap our creation in texture-mapping and program it to duckwalk while humming "In the Mood" through the PC speakers. Technology has provided the artist with a lot of neat tools to create slick effects. But there's no item on the toolbar that'll add life to your picture.

That challenge is yours, whether your artwork is static or animated, whether representative of biomorphic forms, landscape, or hardware, whether realistic, cartoonish, or highly stylized. There is a personality that transcends mere likeness, and this must be searched for in every subject, including those that are inanimate or completely imaginary. The artist's real work begins in seeing, in perceiving those nuances. The rest of the job is just figuring out how to show your audience what you've seen.

Subduing the Superhero

The chief cause of flaccid art, a visual cliché, is the result of the artist making an assumption rather than an observation. A common example is the depiction of masculine muscularity by aspiring artists who got their anatomy training from superhero comics rather than life drawing classes. The various bodily bulges have become codified and stylized and are added like clip-on parts or Colorforms: here's an arm (bulge, bulge, bulge), and it's attached to a torso (six little abdominal bulges, two big lumpen pectoral bulges).

Of course, when depicted in this fashion, the parts have little, if any, relation to one another—unlike a real musculature that can be observed to function as a unit—and the result is recognizable but thoroughly unconvincing. Similar visual clichés abound for virtually every subject matter, and a good artist works to avoid them.

What takes their place is observation; meaning, ideally, that the artist should look at something real and sketch it while it's actually present. A radical concept, I know, and not always easy to accomplish. If you just can't arrange to sketch live models, try working from video or laserdisk or, failing that, from photos. The important thing is to work from something other than your memory or imagination (just for this exercise, not forever).

Try it, and don't be afraid to attempt a subject in motion. Work fast. Go for the general shape of your subject. Sketch contours. Use quick, spare lines to capture the essence of movement, balance, mass. Watch your subject rather than the lines you are making on the page. Work toward suggesting character with an economy of line. Don't worry that it doesn't look like much. For that matter, these sketches should be largely indecipherable to anyone other than the artist.

Being aware of nuances of real-life movement will enable you to imbue on-screen movement with greater authenicity.

This sort of exercise, called gesture drawing, is valuable for any subject matter and worthwhile whether the result is a static or moving image. Gesture drawing helps the artist loosen up, allowing mind and hand to flow together. It also is crucial for distilling the essence of a subject, for capturing that convincing bit of personality that makes a piece really work. Toward that end, your sketches serve as invaluable notes in visual shorthand.

Colorful Background

Usually, backgrounds are executed by a different artist or artists than are the screen's moving elements. Nonetheless, it is important that a certain consistency of style is observed between the two if a satisfying result is to be obtained. A great disparity between foreground figures and background scenery robs a piece of what life it has because the whole is made to appear unconvincingly patched together.

Equally important is a carefully considered color palette, with distinct yet harmonious colors for figures and background. Use of an identical color in foreground and background can cause a conflict if the two areas are ever allowed to overlap on the screen. Even if no overlap occurs, such misuse of color will only serve to detract from the illusion of depth you've worked to create. The colors selected for the background should always work to make it recede, giving precedence to foreground action. An effective way to achieve this is with reduced color saturation (from the mysterious HLS values).

It's Alive

Compared to traditional cel animation, the computer can make it so easy to move an image smoothly across the screen that it is tempting to ignore the fact that few things really move quite that fluidly. This is reflected in an important principle of animation known as anticipation, which refers to the minute shifts of weight that preface a movement, that occur during its course, or that follow in its aftermath.

As an example, watch someone move from a standing position to a brisk walk and come again to a stop. Better yet, pay close attention while you perform these actions yourself. Notice how the body's weight is shifted to initiate the movement, and the first step is not the same as following steps where momentum has been gained. Observe how and when the swinging of the arms comes into play. When forward motion is halted, note that all movement does not stop abruptly. Pay attention to how the body in coming to rest redistributes its weight.

Of course, not all animation has to attempt to reflect this degree of fidelity. By being aware of such nuances of real-life movement, however, the animator will be able to imbue on-screen movement with greater authenticity. This is worthwhile even in less-realistic styles of animation.

A great example of this can be seen in Disney's *Snow White* in the scene where the seven dwarves first appear. Go ahead, laugh. This is not just seven stumpy figures moving their limbs back and forth. The dwarves march in unison, yet each displays a unique personality evident in his walk. Seven walks, all different, all filled with character. The scene is a small masterpiece of expressive animation and should serve as an inspiration to all animators, even those using fancy software to render space battles. Remember, in every subject there is a personality to be discovered and conveyed.

No Gravedigging Required

And you thought the toughest part of mastering computer graphics was going to be getting through the manual. Not quite, but compared to the hardships suffered by Goldblum's Seth Brundle—or a Victor Frankenstein—bringing your game visuals to life will be as easy as falling off a lab bench. True, you've got a good deal of work and planning to do before sitting down at the computer, but at least the torch-brandishing villagers leave you alone, and there are no baboon parts to clean up.

So grab a sketchbook and go sit outside. I'm going to the video store to see if Disney has made *Snow White* available yet. Heigh ho! ■

Despite appearances to the contrary, David Sieks is not dangerously fixated on Snow White *(nor on the Seven Dwarves, nor on any animated character, with the possible exception of Charlie Tuna, for whom he pines terribly). He writes and makes his little artistic dabblings in Boston or thereabouts. Sieks can be reached via e-mail at dsieks@arnarb.harvard.edu or through* Game Developer *magazine.*