gd

GAME DEVELOPER MAGAZINE

OCTOBER/NOVEMBER 1995

UBM Tech

GAME DEVELOPER

The Sound of Jerry

Jerry Lewis breaks character: "Listen...lay-DEEE!" It's not much, just a single word delivered in that characteristic way, but the audience goes nuts. Everyone on stage freezes for probably a good 15 seconds while the Marquis Theatre roars with approval. The septuagenarian has let the audience know that tonight it's not going to be the saintly, Brylcreem-haired, (ex-) chain-smoking, dear-dear-friend-of-Liza, nominated-for-a-Nobel-Peace-Prize, hyper-earnest shill with the 21-hour Labor Day patter. Tonight, it's going to be the zany, wacky goofball with the (once-) rubber skeleton, the Geisha Boy, the Nutty Professor, the Jim-Carrey-Eat-Your-Heart-Out daffy laughmeister. Tonight, we know, there will be shtick.

Jerry Lewis is playing Applegate (aka The Devil) in the Broadway revival of *Damn Yankees!*, and the audience is filled with people with the good sense to turn up their noses at *Beauty and The Beast*, *Sunset Boulevard*, *Cats!*, and half a dozen other spectacles for which stage effects and the sheer shock of live orchestral music played competently stand in for...well, what? The *Damn Yankees!* sets are spare, filled with pastels, forced-perspective, and dressed with what look like authentic chrome-and-bakelite radios, toasters, and card tables. There's some of the stage magic demanded of a Broadway show nowadays, with sets rolling on and off silently, descending from the catwalks, popping up through trapdoors, a shim or two, and so forth, but you get the feeling it's the bare minimum required by the unions. The cast is fabulous both in form (when Charlotte d'Amboise, playing Lola, first appears, I do the whole Tex Avery thing: my eyes pop three feet, smoke shoots from my ears, and I involuntarily scream "Hooga-hooga!") and function, but I'm sure the various Andrew Lloyd Webber spectaculars offer at-least-similar levels of competence.

Damn Yankees! succeeds in plastering a goofy grin on your face for two-and-a-half hours not through any one thing (although the presence of the King of Comedy is obviously what draws a full theater every night), but through a balance of elements that are a lesson to anyone in digital entertainment. The play itself is fluff, even by the standards of 50s musicals. The book is fine but not particularly familiar (the only song you're likely to recognize is "[You gotta' have] Heart."). Indeed, it's the very modesty of the play, coupled with professional delivery, that makes it so enjoyable.

You don't need state-of-the-art spectacle to entertain. Indeed, state-of-the-art spectacle can easily overwhelm a modest entertainment. Although I know I'm in the minority, I've always felt that Doom was inferior to Castle Wolfenstein 3-D, its technically limited predecessor. That's one reason I was doubly delighted when the programming wizards at id made the source code for Wolfenstein available for public perusal (you can download it from our ftp site or CompuServe forum; the filename is WOLF3DSC.ZIP).

What you need is the enthusiasm and courage to make the game you want to make. A little talent helps, but it's not as important as heart. When others say you can't win, that's when the grin should start. You gotta have heart... Oops. There I go with that damnably hummable tune from the show again. ■

Larry O'Brien
Editor

Here is the final verdict on accessing *Game Developer* code. Go to our ftp site at ftp://ftp.mfi.com/gamedev/pub/src. If you're using Compuserve, GO SDFORUM. That's it. We promise.

A 3D Benchmark for the Industry

Alex Dunne

Creating a method to game play madness may help consumers decide what games will work best on their systems. Development of this industry-wide standard won't be easy—will you be involved?

Recently, I began receiving a steady stream of e-mail from Ken Nicholson of ATI Technologies. Ken's an active member of the GamePC Consortium, a group of close to 100 member companies, each of which has a hand in the game development arena, including title developers, tool vendors, system manufacturers, chip, and board manufacturers. The stated goal of the consortium is a simple and noble one: to raise the level of game play on IBM PCs and compatibles.

The e-mail that Ken sent consisted of discussion threads about two related issues that are long overdue for addressing: a three-dimensional graphics benchmark and a hardware specification for PC game play.

The intent of both issues is to provide consumers with a way to look at a game on the store shelf and tell whether the title will run adequately on their system. It's far from an easy task to get an entire industry (or even a portion!) to agree upon sticky issues like these, but if a middle ground is reached, it will be a huge leap forward for consumers and developers alike.

This month, I'm going to focus on the consortium's proposal for an industry-wide three-dimensional benchmark that would serve as one of the components within their second aim, the GamePC hardware specification. Like the Multimedia PC Working Group's MPC specification for multimedia hardware, the GamePC Consortium wants to create a symbol or trademark that would indicate the level of hardware required for different games.

Although the ideas are still being bantered around, I imagine the rating system would be two or three tiered, each tier representing a computer with progressively more power and capability. Using this rating system on game boxes would instantly indicate to consumers the type of system required to enjoy the game. A rating of this sort necessitates the hardware manufacturers getting involved because it's their products that will be rated. Many companies have jumped on board already.

The hurdles to developing a three-dimensional benchmark, as you might surmise, are fairly large. Benchmarks are notoriously hard to devise and maintain, even when you don't have to reach a consensus across the industry. The idea of using Microsoft's Funstones benchmark (which isn't available yet) or asking an independent organization such as Ziff (the folks at *PC Week* Labs) to develop a benchmark been discussed. Although either of these scenarios may come true, right now the three-dimensional benchmark development is in the hands of the industry–of which you are a part.

The discussions by consortium members about the benchmark have been very insightful. Although this article represents an excerpt of the discussion, it gives you an idea of the problems that have to be solved before devising a standard three-dimensional benchmark.

Define What's Being Tested

First, Rob Glidden of SoftPress cut right to the question that was rolling around in my head: "What are you testing: the 3D API, the chip, the board, the host processor, the device driver, or a 3D application? All these are in the pipeline for getting a 3D image to the screen. You need a way to isolate out the relative performance contribution of one aspect. To do this, you need a baseline and a flexible enough implementation so that no one is left out."

Glidden proposed finding one or more games that run on a number of different chip sets and using the games' frame rates on each configuration as an indicator of performance. He then explained the drawback: every chip maker would have to make sure that those particular games ran well on their chip sets. There's also the problem of finding the proper game to exercise all facets of the hardware within a short enough time without having to jump to different scenes, levels, or whatever to get the right mix of animations.

Glidden also submitted ideas for measuring polygons per second, pixels per second, and frames per second. Each, he explained, has its Achilles heel as well: What size polygons, and how many vertices? Are pixel fill numbers too abstract for consumers? Frames per second containing what kind of objects? These questions were insightful, and Glidden's final comment was also good: there's a qualitative element that cannot be objectively measured by a benchmark. For instance, one chip set that blows away another on a frame rate benchmark displays inaccurate colors. Which is better in that case?

Another person who echoed the concerns of basing a three-dimensional benchmark on objective vs. subjective yardsticks was Dan Wood of Matrox. He summed it up like this: "A word of warning—testing three-dimensional accelerators will be very difficult, as most hardware...will offer different levels of quality and features besides speed. In some cases, boards may be able to achieve very high frame-per-second rates, while rendering very low quality. Unfortunately, image quality is a very difficult thing to account for in benchmarking." It's a problem that I certainly don't see any easy way around, short of appointing someone or a group to subjectively judge the quality of rendered images. And that's opening a whole new can of worms.

Coping with the Three-Dimensional API

To account for the effect of the three-dimensional API on hardware tests, there were a number of suggestions:

- Everyone could agree to standardize on one API for the testing. Dan Wood as well as Lou Long of Oki Advanced Products suggested using Microsoft's 3D DDI or Intel's 3DR. However, this may have to wait until the day when either of these APIs is in widespread use. An interim solution would have to be found until that day comes, if ever.
- The group could optimize the benchmark for each API. However, that situation would be quite a chore for the consortium (which is young and probably not too flush with cash). Because the benchmark would have to be optimized for every API, the consortium would have to maintain multiple benchmarks over a period of time as the APIs evolved.
- The group could devise the benchmark to be API independent. In my mind, this seems to be the best option.

To illustrate how an API-free benchmark could be accomplished, Neil Trevett of 3DLabs proposed a test of the whole system, analogous to Glidden's idea. He would devise a realistic test, one that required moving overlapping three-dimensional objects, hidden surface removal, and back-face culling in front of a textured background (he suggested multiple spheres, cubes, and so on, bouncing around and colliding in a three-dimensional box).

The benchmark would specify what textures to use, the lighting, the position of the eye, and other details. Then, when the benchmark is tightly specified, the source and binaries would be distributed to the various

WHO'S INVOLVED?

If you are interested in finding out more about the GamePC Consortium or want membership information, the group can be reached at:

- **http://www.mmwire.com/gamepc/gpchome.html**
- **CompuServe: GO GAMEPC**
- **or contact Ken Nicholson of ATI Technologies at 75300.2772@compuserve.com.**

Here's an sampling of GamePC Consortium member companies:

GAME DEVELOPERS:
Acclaim
Accolade
Broderbund
Interplay
Lucas Arts
Maxis
Microprose
Origin
Spectrum Holobyte

TOOL VENDORS:
Argonaut
Autodesk
Caligari
Intel

SYSTEM VENDORS:
Acer
Apple
IBM

CHIP AND BOARD MANUFACTURERS:
ATI
Cirrus Logic
Creative Labs
Diamond Multimedia
Matrox
Media Vision
Trident
VLSI
Weitek
Yamaha

hardware companies, and they could use whatever API they wished for the testing, as long as the resulting animation was identical to the specification. This would also mimic the optimizations that game developers make for various hardware.

Oh Yeah–the Other Platforms

While virtually everyone involved in the discussion was focused on Windows 95 as the testing platform, Dave Thielen of Windward Studios reminded everyone that there were other operating systems and environments to consider.

"This benchmark needs to run on Win16, Win/NT, and OS/2 as well as Windows 95," Thielen said. "In all the hoopla over Windows 95, everyone seems to have forgotten that Win16 still owns the desktop market and OS/2 has 10%–while Windows 95 has 0%." Very good point, and one I hope people don't overlook. As I mentioned last time ("The Windows 95 Game Plan," August/September 1995), the market for Windows 95 is not going to kill the DOS and Win16 game market overnight, and it bears some consideration in whatever three-dimensional benchmark is adopted.

Testing Conditions

Once the GamePC Consortium agrees on a standard benchmark, whether it's API independent or dependent, it must specify the conditions under which the benchmark will be run. The tests must occur in an environment every company can duplicate, and none can deviate from in the interests of a better performance rating. Lou Long explained that the test "...must state the physical parameters like the width and height of the window (if not full screen) in pixels, the depth (bits per pixel), the color mode (palletized or true color), and any other parameter that affects the total number of bits rendered by the hardware."

So, as you can see, a flock of issues must be dealt with before a PC three-dimensional benchmark is finalized. Despite the technical hurdles, I believe the benchmark will be here sooner rather than later. If you'd like to throw your two cents into the discussion, I've included the contact information for the GamePC Consortium. You may be testing your own games with this benchmark in the near future, so it's worth the time to donate input. Next time I'll examine the plans for a GamePC hardware level; it is akin to MPC hardware ratings of which the three-dimensional benchmark is but a single part. ■

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

Sweet Animation Suite

Nicole Claro
Barbara Hanscome

Sound, animation, upgrades, trade shows, and gossip. These are the tools of the trade. Here's a sampling of some new and upcoming stuff you might want to check out.

Even though you shouldn't put all your eggs in one basket, count your chickens before they hatch, or put the old cart before the horse, you should try to keep all your development tools in one place. What better way to expedite three-dimensional animation than with a full suite of tools at your fingertips?

Crystal graphics is now shipping just such a product. The company's new Crystal Kaleidoscope (pictured here however minutely) is centered around

CrystalGraphics's Kaleidoscope features a quartet of powerful rendering and animation tools built around the core of the company's TOPAS technology.

TOPAS Professional 5.1 a three-dimensional modeling, rendering, and animation package for the PC. Also included are Fractal Design Painter 3.1 (to create natural-looking art from calligraphy to oils to airbrushes), Kai's Power Tools 2.0 SE (gradients, textures, and filters), Elastic Reality 1.01 (for morphing and special effects), and Leadview 3.0 (for managing and compressing images). Each tool has won awards on its own. In addition, the set comes with four CD-ROMS with textures, images, and three-dimensional models from a variety of other manufacturers. Crystal Kaleidoscope retails for $1,995.

- For more information contact:
  CrystalGraphics Inc.
  3110 Patrick Henry Dr.
  Santa Clara, Calif. 95054
  Tel: (408) 496-6175
  Fax: (408) 496-0970

Bring the Noise

Now you can perform a post-recording cleansing on any game audio you create. Tracer Technologies recently began shipping its Digital Audio Reconstruction Technology (DART) software. DART removes all surface noise, pops, clicks, and other disturbances from any audio source. As long as you're using a Windows-compatible sound card, you can use DART.

After recording, the software applies TriCleanse, a three-part process that includes a smoothing processor, a postfiltering processor, and an outlier detector. The first component smooths and reconstructs the signal, the second removes surface noise, hiss, and any other distortion, the third searches for hard noises, such as pops and clicks, and automatically removes them. You can even keep the noise you've removed in a separate file and reapply different levels of the TriCleanse process until you're satisfied with the sound quality. DART also features Soundtree, which lets you review several takes of a sound file and choose your favorite one. Its toolbox includes an eight-band graphic equalizer; cut, copy, and paste editing; sound left

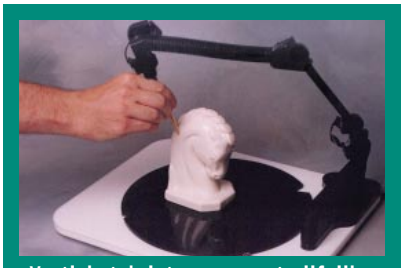
and right splitter; low, high, bandpass, and notch filtering; visual and audio markers; unlimited active windows; a gain adjuster; and a mixer application. It retails for $399.

■ For more information contact:
Tracer Technologies Inc.
P.O. Box 188
Dallastown, Pa. 17313
Tel: (717) 747-0200
Fax: (717) 741-6709

Digitize, Digitize, Digitize!

Looking to make realistic, three-dimensional models in less time? Check out Vertisketch 2.0, a new plug-in from Idaho's Blevins enterprises. (I spent two days in Boise on a recent road trip. I am now officially a seasoned traveler.)

Vertisketch 2.0 is designed for use with LightWave 3D 4.0 and any supported three-dimensional digitizer. It features many new functions such as lofting, which automatically creates



Vertisketch lets you create lifelike three-dimensional models with the simple stroke of a digitizing brush.

polygons, and autosurf, which creates a surface at the stroke of a key. Vertisketch 2.0 comes in two different models, one that relies on a footswitch alone and one that incorporates a rotating digitizing platform.

■ For more information contact:
Blevins Enterprises
121 Sweet Ave.
Moscow, Id. 83843
Tel: (208) 885-3805
Fax: (208) 885-3803

Two-Legged Wonders

To accompany its upcoming 3D Studio MAX (the newest version of 3D Studio for Windows NT), Autodesk has announced Biped, a software product for creating lifelike, free-form animations of two-legged characters.

According to Autodesk, the technologically advanced software combines several innovative techniques with advanced inverse kinematics and skinning functions which let the characters move at any pace. Gait, arm swing, and center of gravity will automatically correspond to any pace you create. You use Biped by creating a set of character footprints, which you then manipulate easily. Change the position and time on the ground of one footprint from the next, and your creation can walk, run, stagger, dance, jump, flip, or any variety of these movements. Biped features three elements that allow for this: step-driven animation, free-form keyframing, and physics-based interpolation. It also includes a skeletal/skin deformation module for creation of seamless joints and realistic muscles and body mass changes as a character moves over time. Biped and 3D Studio MAX will ship sometime in 1996.

■ For more information contact:
Autodesk Inc.
111 McInnis Pkwy.
San Rafael, Calif. 94903
Tel: (415) 507-6112
Fax: (415) 507-6112

Schmooze News...

Christmas Crunch Sneak Attack...As we move into Christmas crunch season id has, with one master stroke, subverted the competition's programmers by releasing the source code for Wolfenstein 3-D. What game programmer could resist delving into John Carmack's code to the detriment of his or her current deadline? The source is available on the Internet from ftp.idsoftware.com or on CompuServe in the GAMDEV library. There's an intro by John Carmack pointing out what's cool and what needs updating.

Shareware Authors out for blood...Game authors are on the warpath against unscrupulous publishers who take shareware to retail and CD-ROMs without permission. Apogee, id, MVP, Redwood Games, and TriSoft are all sicc-ing lawyers on these pirates. With the entry of Interplay's Descent into the shareware market, we may see some expert copyright muscle behind this effort. Question: What giant distributor/publisher told a developer whose game it had pirated, "We're so big we can use our petty cash fund to keep you tied up fighting us until you go bankrupt"?

Musical Chairs...Matt Gruson has joined Disney Interactive because he "saw a lot of interesting opportunities and the chance to work under an experienced management team." Gruson (whose first big game was Earthrise back in the 80s) founded the Graphic Adventure Group at Microprose, developed the MADS game engine which was used in Rex Nebular (still on the top 100 list of games), and then went to Sanctuary Woods. He says the Graphic Adventure Group was "the most energetic and wonderful group," and his goal is to get together with a group like that again. When asked if he'll try to reassemble the crew (some of whom followed him to Sanctuary), he replied, "No comment."

Sega Exodus...Wallace Poulter has left Sega to become an Executive Producer of GameTek's new sports division. In a switch, instead of Poulter having to move to Florida, GameTek will move to the San Francisco Bay Area. Relocating all those Sega folks they picked up must have looked more expensive than moving the whole company. Others jumping from Sega to GameTek include Tom Reuterdahl (now VP of development) and Andy Johnson, who will work under Poulter. Meanwhile the exodus from Sega continues, with Chris Garske going to Good Times Software and Wayne Townsend, former head of Sega Sports, leaving to start his own development group.

Got gossip? E-mail The Gossip Lady *at 71501.3553@compuserve.com.*

Kids are People, Too

Creators of children's edutainment will have the chance to connect with 3,000 of their potential customers October 5-8 at the second Children's Multimedia Expo in San Francisco, Calif. The event, which debuted in June of 1995, will bring together thousands of kids, educators, parents, retail buyers, and multimedia pros for four days of show and tell. Edutainment developers will have the opportunity to gain important feedback on their products during demos and focus group testing with some 3,000 children and teachers brought in from schools throughout California. (Wow kids, here's a field trip to a place where you can play computer games all day long—the only catch is you've got to answer a few questions from some friendly marketing executives....) Last year, multimedia companies demonstrated works in progress as well as titles currently shipping.

Two days of the four-day event are closed to the general public. During the weekend, the doors open to everyone and special family-focused Internet and World Wide Web demos and workshops will take place. At the end of the Expo, students and teachers will vote on their favorite titles in the Kids' Choice and Teachers' Choice Awards.

Vendors exhibiting at the show are by invitation only. Companies interested in participating in this event or future Children's Multimedia Expos must submit their products for review. For more information, contact event producer Shannon Tobin at (415) 788-9990 or fax (415) 243-8939. ■

■ For more information contact:
Shannon Tobin
Event Producer
Children's Multimedia Expo
Tel: (415)788-9990

Nicole Claro is managing editor for Game Developer *magazine. Barbara Hanscome is managing editor for* Software Development *magazine.*

UPGRADE YOURS!

- Caligari's newest upgrade is trueSpace2 for Windows. It features new animation and video capabilities and trueClips, a CD-ROM of more than 200 textures and 600 three-dimensional objects. The upgrade is available to trueSpace1.0 users for $149 and $795 for new users.
- Borland is shipping Turbo C++ 4.5, which includes five free games with full source code. This latest version is geared toward beginning programmers and retails for $79.95 for new users and $49.95 for owners of any previous versions of Turbo C++.

Memory Miscellanea

Chris Hecker

Don't worry, perspective texture mapping is alive and well. But this month, Chris Hecker takes a break from his series on texture mapping to explore the nuances of memory bandwidth in game programming.

As you can see from the title, this article is not "Perspective Texture Mapping, Part IV," the continuation of our epic perspective texture mapping series. Don't panic, I'm not breaking my promise to deliver a wicked fast perspective texture mapper, but to break up the series a bit I thought I'd insert a non-texture mapping article here in the middle (although the topic is definitely applicable to texture mapping, as you'll see). We'll resume with Part IV next issue.

This time through, we're going to discuss memory bandwidth. Plainly stated, memory bandwidth is a measure of how much memory you can read and or write in a given amount of time.

From that description, it should be clear that memory bandwidth affects every kind of game on every platform, from scrolling platform games on an 8-bit Nintendo or Atari 2600 system to high-end military simulators that cost millions of dollars. Memory bandwidth governs how many sprites the hardware in the older consoles can move around, and how many polygons can be textured per second in hardware on the newest machines or in software on the PC.

In fact, an oft-cited goal of PC graphics programmers is to "get your texture mapper running at memory bandwidth," because there's not much more you can do to increase its speed after that. To get a tad flowery, memory bandwidth can be an open door or a brick wall. It all depends on how much of it you've got.

Lies, Damn Lies, and Bandwidth Numbers

On today's machines, we usually measure memory bandwidth in megabytes per second (MB/s), but you'll sometimes see bytes per second, dwords (a dword is four bytes in this article) per second, and so on. If you're looking at a bunch of memory bandwidth numbers, it's obviously important to know which measurement units they're in.

Like most statistics, the bandwidth numbers themselves aren't enough to tell the whole story, and you need to know exactly how the numbers were generated for a specific machine to give them meaning. For example, I could tell you the laptop on which I'm typing right now gets 42 MB/s, but you really aren't any more knowledgeable than before because you don't know if I mean read bandwidth, write bandwidth, copy bandwidth, sequential or random reads or writes, or any combination thereof. All these parameters can make a big difference inwhat a bandwidth number really means.

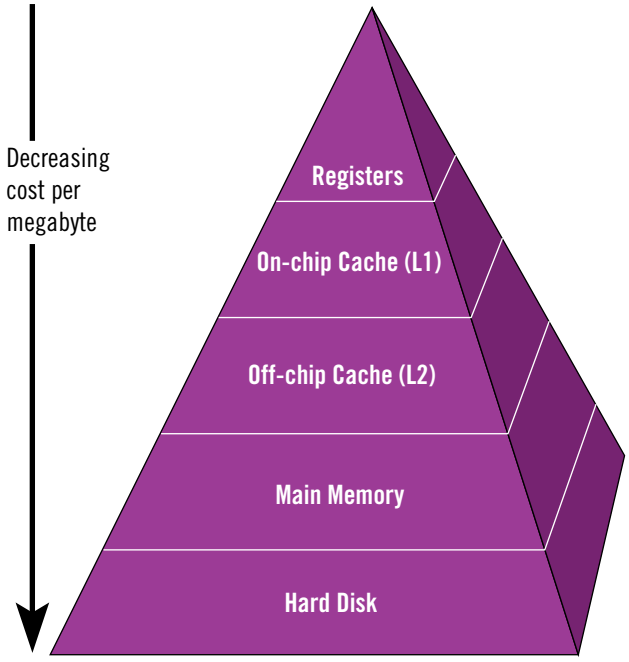
In fact, it's rare that any general bandwidth number will mean anything in the context of your specific game. I'm going to talk about various things that can affect your game's memory bandwidth, techniques for measuring that bandwidth, and pitfalls you'll encounter along the way.

Pyramid Power

First, we need a one-minute refresher on how modern CPUs and motherboards work. I'm certainly no hardware engineer, so we'll limit our discussion to how the software sees the hardware.

Throughout computer history, there's always been a pyramid diagram

Figure 1. Component Cost vs. Amount



that describes the speed of the component vs. how much of that component you're likely to have in your system (usually because faster components are more expensive). Figure 1 shows this diagram for memory components.

At the top of Figure 1 we have the CPU registers, which usually take a single cycle to access and are the most flexible of all types of memory, but are pretty scarce (Intel x86 processors have only eight general purpose registers, each of which holds four bytes, while most new processors like the PowerPC have around 32—all the parenthetical numbers in this paragraph are estimates and will vary in practice). Below the registers, we have the on-chip cache memory, sometimes called the level 1 cache. This is usually a small amount (8 to 32KB or so) of fast memory with access times slightly slower than registers. Cache memory is a little less flexible than registers, as well. Most CPU architectures don't let you add a number in the cache directly to another number in the cache without using the registers for temporary storage.

On the next rung down, we have the off-chip level 2 cache, which usually has more storage space (256KB to 1MB), but is much slower than the on-chip cache, generally on the order of five times slower or more.

Second to last in our diagram, we have main memory—of which there's usually a relatively large amount (4 to 32MB). As you'd expect, it's even slower than any of the memories above it. Finally, we end up with the hard disk, which has oodles of storage (well, okay, my hard disk sometimes has less space free than I have main memory, but it still has more raw storage!) if you're willing to pay for the access time and transfer rates. CD-ROMs and tape drives would be below hard disks if we put them in the diagrams, because they're cheaper (and slower) per megabyte.

The most interesting thing about Figure 1 is that it holds for almost every machine architecture, from Commodore 64s to Crays. On the lowest end, you might not have caches, and at the higher end you might have more layers, but the speed vs. cost ratios still stand.

With that refresher, let's get to the hints, tips, and techniques for determining the memory bandwidth for your game, so you can strive to achieve it.

What's Your Access Pattern?

As I mentioned, a single number doesn't tell the whole story about memory bandwidth. In fact, there are zillions of different kinds of memory bandwidth, each different because the access pattern used to generate the numbers is different. The access pattern is the way the application moves the memory around, and there are as many different types as there are programs. Three general categories that are important, but by no means form a complete list, are sequential copy bandwidth, sequential write bandwidth, and random read-sequential write bandwidth.

Sequential copy bandwidth is the number that applies when you're copying a block of memory from one place to another—to copy a new piece of digital audio into the play buffer, for example. Sequential write bandwidth is what you see when filling a rectangle or polygon, zeroing an array, or anything else where you're writing a single value or a value that's generated using instructions (as opposed to read from a source) to a destination. Finally, random read-sequential write bandwidth is what you see when texture mapping, where your source locations are fairly randomly distributed, but you're usually writing a scanline at a time to the destination.

From these descriptions, you can easily come up with other kinds of bandwidths and situations in which they'd arise. You might find multiple reads and a single write interesting if you're mixing digital audio or alpha blending sprites. Likewise, a single read and multiple writes might be your thing if you're stretching an image. The key is to figure out which type of bandwidth is most appropriate to your application and measure it.

It's clear that any useful and interesting application is going to do more than just copy bits all day, but memory bandwidth gives a good upper bound on your performance. In other words, even if you were the best optimizer on earth, you still wouldn't be able to get your code faster than memory bandwidth if you need to move those bits around. This may seem like a limitation, and it is, but you can also look at it as an opportunity. If you can figure out a way to reduce your memory bandwidth requirements by redesigning your algorithm or possibly by changing your

access pattern you can open up new possibilities for optimization.

An Accessible Example

For example, if you were writing a solid polygon rasterizer, you could measure sequential write bandwidth and compare it to the fill rate you get through your rasterizer. The difference is your overhead above memory bandwidth. Your goal is to minimize this overhead or, if possible, cheat somehow so you get the same effect on the screen but aren't bound by the same memory bandwidth limitations.

Let's say you have the world's fastest 90-degree bitmap rotator; you can take a bitmap and rotate it 90 degrees at memory bandwidth on your machine—you're very proud of this code. You know it works at memory bandwidth because you measured it without any instructions except the copies in the inner loop and got the same bandwidth number when you added your rotator code. Let's also say your code is "destination-centric," that is, it scans horizontally in the destination and therefore it scans vertically in the source to accomplish the rotation. Of course, you measured memory bandwidth doing the same thing, so let's call this access pattern vertical read-sequential write. Since we're running up against this memory bandwidth limitation, how can we restructure the algorithm to have a different limitation?

It's immediately apparent that you should measure sequential read-vertical write, which will accomplish the same rotation, but might be a different speed. Also, another possibility is to spend a little memory and pre-rotate your bitmaps, so your access pattern is sequential copy. Will either of these be faster? I don't know, and we can't say with certainty until we've timed it. My hunch is that sequential copy will be the fastest in terms of pure bandwidth because it's probably the access pattern for which the memory subsystem was optimized, but it's just that, a hunch. It's entirely possible the extra memory overhead from pre-rotated bitmaps would make the overall code slower because of paging.

The real solution, if this is a bottleneck in your game's run-time speed (and you shouldn't even be bothering to measure this stuff if it isn't a bottleneck), is to profile the various techniques at startup and have your game self-configure to use the fastest possible pattern for the given machine.

What's fastest on your machine might not be fastest on mine. The best we can do is have a list of things to look for when we're measuring bandwidth.

It may seem like I'm being wishy-washy by not just declaring a single access pattern the fastest, but we've got far too many variables to do so. The problem is compounded by the number of different hardware architectures out there, so what's fastest on your machine might not be fastest on mine and vice versa. The best we can do is have a list of things to look out for when we're measuring bandwidth. Cache effects would definitely be at the top of this list.

Understand the Cache

The processor cache is usually an object of great fear, wonder, and misunderstanding. A friend of mine named Terje Mathisen says, "All programming can be thought of as an exercise in caching." Although Terje isn't talking specifically about the processor cache, this is a rule to live by when you're trying to optimize on modern processors. If we apply this idea to the processor cache and memory bandwidth, it means, "Figure out how to put your important data in the cache and keep it there." This may seem obvious, but keeping your data in the cache is more difficult than you might think.

Before we bother getting into this, what difference does it make? Well, on my laptop, the speed difference between reading from the on-chip cache and reading from sequential uncached memory is tenfold—and this isn't even the whole story. I'm reading sequentially in this example, so at least some of the reads are cached for reasons I'll explain shortly. If I ensure that all the reads are uncached by reading pseudo-randomly, the program reads from the cache about 30 times faster than from main memory. You can do a lot in 30 cycles on a modern processor, so I'd rather not spend them waiting on memory.

I'm going to assume you know generally what a cache is and how it works, so the only high-level description I'll give is this: the cache stores frequently accessed data in fast on-chip memory, so when you reference it the chip doesn't have to go out to the memory bus to fetch your request.

Caches are broken up into cache lines, which are usually 16 or 32 bytes long, and the processor reads in an entire cache line from memory when a cache miss occurs. These cache lines are aligned on address boundaries that correspond to their length, so 16-byte cache lines are aligned on 16-byte boundaries, for example (addresses ending in 0 hexadecimal). This is why my previous sequential reads were partially cached. Assuming a 16-byte cache line, every fourth dword I read in my test brought in another cache line,

and the next three dwords were read from the cache's fast memory. The use of cache lines also means that if you're referencing two bytes at addresses that differ by more than a cache line (or if there's a cache line boundary between them) you'll be using two lines, even if you're only accessing those two bytes.

Life in the cache gets even worse when we delve deeper into its behavior. Most modern caches are N-way set associative for some small integer N, usually 2 or 4. A cache set is a group of N cache lines, so a two-way set associative cache has a bunch of sets, each containing two cache lines. You can tell how many sets there are by taking the size of the cache in bytes, dividing by the number of bytes per line to get the total number of lines, and then dividing by N for your cache to get the total number of sets. For example, the Pentium has 8KB of two-way set associative data cache with 32-byte cache lines, so 8KB / 32 bytes / 2 lines per set = 128 sets.

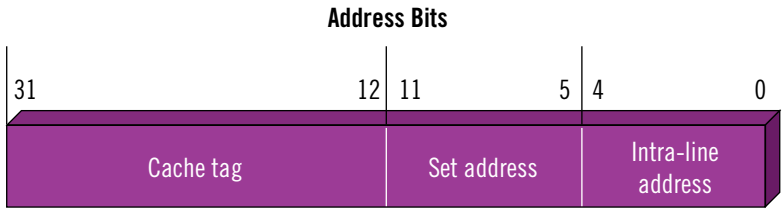
The cache translates a memory address into a cache line address by using the lowest bits for the intra-line address, the next few bits for the set address, and the remaining high bits for the cache tag. Figure 2 shows the breakdown for the Pentium. Because there are 32 bytes in a cache line, the lower five bits are used for the intra-cache line address, and the next seven bits give us the 128 sets we calculated previously. Which line a given address uses in its set is up to a replacement algorithm (usually least recently used or an algorithm close to it) based on the cache tag bits. In other words, every address that contains the same set address bits will map to the same set, and all those addresses must share the lines in that set.

This is where the replacement algorithm comes in. If all the lines in the set are currently full and none of the tags matches the requested address, then one of the currently cached lines needs to be replaced by the current requested line. For example, on the Pentium only two of the many possible cache lines (20 bits worth of lines because bits 12-31 make up the tag—that's 1,048,576 possible lines!) for a given set can be in the cache at the same time.

You can see why this works well in the general case because referenced addresses are likely to be near each other—a phenomenon called locality of reference—so they'll have different set addresses. The set architecture allows for some addresses to be not-so-near each other because it lets very different addresses with the same low bits map to N different cache lines (in contrast, a cache architecture called direct mapped has no sets, so each address with the same low bits shares a single cache line).

However, in certain cases this kind of cache architecture can really screw you up. For example, let's say you're reading vertical strips from a bitmap like the bitmap rotator we discussed previously—down one vertical scanline, then down the next, and so on. The width of your bitmap will dictate how much of the cache you end up using. If your bitmap is 256 bytes wide, a single increment vertically will step bit 8—and never any bits below bit 8—in your address. If you compare that with the Pentium's cache address layout in Figure 2, you'll notice that you're only using four bits of your possible seven bits of set address. This means that instead of using all 128 sets, you're only using 16 of them or only 1/8 of your total cache! The startling implication of this is the next horizontal byte from the first scanline will not be in the cache when you get back up there if you've gone farther than 32 scan lines (16 sets x 2 lines per set) because it's been pushed out of its set by another line.

Figure 2. Pentium Cache Addressing



Assume Nothing

What can you do about this sort of thing? Well, first you need to realize when it is happening in your code. To do this, you need to get good at profiling. Profiling at this level doesn't mean just running the code profiler that comes with your compiler and examining the results, it means figuring out exactly where your code is spending its time in the inner loop. You can definitely use the high level profiler to find the inner loop in the first place, but once you've found it, if you want to max it out and really pin down why it's taking the time it is, you're going to need to get down and dirty with a very accurate timer (personally, I use `timeGetTime` or `QueryPerformanceCounter` on Windows, but any accurate timer will work) and a knowledge of assembly language.

Before continuing I should stress that this kind of profiling and optimization takes a very long time, so you should make absolutely sure you're applying that time to the right part of your code. In a game, there are probably 10 lines of code in the entire project that might need this sort of attention, and if you're going to spend a week looking at them you had better make sure they're the right 10 lines.

Michael Abrash's phrase, "Assume nothing!" and its corollary, "Time everything," are words to live by in this neck of the woods. Abrash is the master of this sort of optimizing, so you should definitley read his book *Zen of Code Optimization* (Coriolis, 1994). As a bonus, the disk that accopanies the book comes with a very accurate timer designed specifically for this in-depth profiling.

While I was writing this column, the need for this very kind of profiling came up. I was gathering bandwidth

statistics for various access patterns on my 486 laptop and I was trying to time cached reads. I know from both experience and the 486 manual that a read from the cache is a single cycle, but I couldn't seem to convince my timing program of this fact. It kept returning around 1.5 cycles per read, and when you're timing at this level that's 50% off. My test program had an unrolled loop of a couple of hundred reads, and then I looped back to the top a bunch of times. I was very careful not to unroll my loop so much that it blew out the code cache, so I simply couldn't figure out what was going on. I timed other single cycle instructions with the same timing harness (using a millisecond timer and looping a lot), and they returned reasonable times, like 1.02 cycles, but my reads kept returning 1.5. If I stuck a nop in between the unrolled reads I got the expected two cycles, one for the read and one for the nop. I stared at the code, trying to find an address generation interlock, (AGI—Intel-speak for a type of pipeline stall), but there weren't any.

Finally it hit me. I remembered that if you're continuously reading from cached memory without allowing even a single free memory cycle for prefetching instructions, the 486 will stall your code to fill the prefetch queue. Eureka! I verified this was the culprit by changing the number of consecutive reads and got the expected one cycle per read. I also looked it up in the 486 Programmer's Reference Manual from Intel, and the stall was listed there among the others.

Time to Cache Out

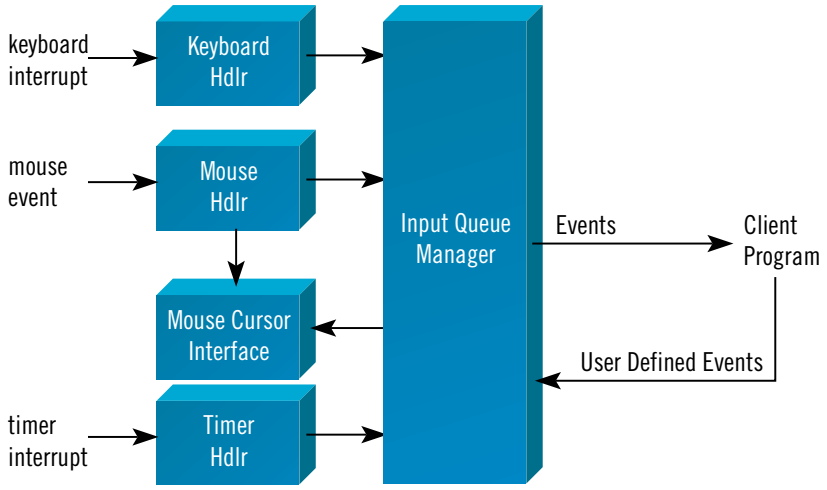
As you can see, figuring out where every cycle is going in your inner loop, especially when there are strange effects brought on by your memory access pattern, is very difficult and time consuming. I highly recommend reading and rereading the manual for your processor before you try to do this. Also, always test your timing program with known inputs so you can verify that it works; Heisenberg is alive and well at this level.

I haven't covered video memory and its associated bandwidth weirdnesses at all. Nor have I discussed processor write buffers, write-back versus write-through caches, processors that don't write allocate cache lines (like the Pentium), new trends in memory that affect the bandwidth numbers (like EDO memory, RAMBUS, and SDRAM), groovy new cache/access pattern debugging instructions (like RDMSR on the Pentium), and much more. Hopefully this article gives you enough background and forewarning about the strangeness you'll encounter that when it's time to max out your inner loop, memory bandwidth won't be the mystery it can be for the unprepared. ■

Chris Hecker wonders why five-year-old workstations with incredibly slow CPUS still have better memory bandwidth than today's top-of-the-line PCs. You can commiserate with him at checker@bix.com.

Organizing User Input, Part I: The Input Queue Manager and Keyboard Events

Figure 1. Input Queue Manager Block Diagram



Most programs I've written have required some form of input from the user. In most cases, the programming language or the library provided with the language was sufficient to get the input in a reasonable manner. Of course, none of these programs was a game.

Fast action games need to process multiple, simultaneous key presses at a frantic rate. Most fighting games provide combinations of keystrokes that, when pressed within a limited time frame, cause some special move to be executed. Not to mention the mouse and joystick, other common game input devices that aren't even supported by most compiler libraries. The question is, how can we obtain the user's input in a reasonably efficient and general manner?

The input queue manager presented in this article is my solution to this problem. This manager accepts the raw input from the hardware devices and stores each input as an event on an internally maintained queue. When a client program requests input, the oldest event is returned.

The main design goals for the input queue manager were:

- Gather user input asynchronously to the client program
- Minimize memory and execution overhead
- Return user input in a single format regardless of the input device.

I developed the input queue manager in C and assembly language using ancient versions of Borland C/C++ and Turbo Assembler, though I've now converted most of the assembly language to C for the sake of maintenance.

The Input Queue Manager Structure

Figure 1 is a block diagram of the input queue manager. The keyboard, mouse, and timer generate interrupts when they are in need of service. The input queue manager handles these interrupts, storing the information gathered on an internal queue and supplying that information to the client program, on request, as events. In addition, the client program may post events to the input queue if required. The mouse cursor interface allows the event manager to accept mouse events even if the video mode isn't supported by the default mouse driver (as in Mode X and VESA).

"Why no joystick handler?" you may ask. The input queue manager only handles events that can be gathered via interrupt. One of my design goals was to provide as little overhead to the client program as possible. The joystick has to be polled at regular intervals for its current position and

button state, so I chose not to make joystick support intrinsic to the design.

Using the Input Queue Manager

Before digging into the nuts and bolts of the input queue manager, I'll present a simple example of its use to give you a feeling for the interface. This example exits after the escape key is pressed. Pressing any other key causes that key's ASCII value to be printed. Listing 1 contains the code for the example.

You must include the header file, inputq.h, at the top of all source files that make use of the input queue manager. This file defines the input queue manager's data structures and provides prototypes for all its visible functions, the names of which are prefaced with `INPQ_`.

The first call to the input queue manager is `INPQ_allocate()`. This routine allocates all required events and a corresponding queue large enough to hold them. The number of events to allocate is supplied as an input parameter to the call.

Next, we enable key press events with a call to `INPQ_enable_keyboard()`. This routine installs a keyboard interrupt handler and tells it to notify us every time a key is pressed.

At this point, every key pressed will enqueue an event. These events will sit patiently on the input queue until the client program makes a call to `INPQ_dequeue()`.

`INPQ_dequeue()` is implemented like the standard C library `time()` routine. If no events are on the queue, `NULL` is returned. If there is an event, a pointer to the event is returned. If an event structure is passed into `INPQ_dequeue()`, it will be used as the repository for the event, and its pointer will be the return value. Otherwise, an event structure internal to the input queue manager will be used. This structure is only valid until the next call to `INPQ_dequeue()` with a `NULL` event parameter.

When `INPQ_dequeue()` is called, the next event matching the supplied event mask, `key_down` in this case, will be removed from the input queue and returned to the client program.

In our example, if the event is valid, it will be checked to determine if the key pressed was the escape key. If it is, we break out of the input gathering loop. Otherwise, the ASCII value of the key is printed.

Finally, `INPQ_release()` is called to deallocate memory held by the input queue manager and restore the vectors of all interrupts that had been taken over. This call is just a formality, though. Because the execution of this routine is critical to the continued operation of the machine after the client program terminates, `INPQ_allocate()` installs it as an exit handler. So, while calling this routine is a good practice, it's not absolutely necessary.

This is all I'm going to say for now about the visible interface to the input queue manager. These functions, along with the functions to enable the mouse and timer handlers, make up the crux of the interface. Table 1 contains a list of all visible input queue manager functions.

The Event Structure

Figure 2 is a graphical representation of the event structure. Listing 2 shows the actual type definitions. Each event is identified by its type field, which contains a

Mike Michaels

Fire keys, key combinations, timed key combinations... What's a programmer to do? Create a single, high-performance input manager, that's what!

Listing 1. Input Queue Manager

```
#include <stdio.h>
#include "inputq.h"
int main (void)
{
    EVENT event;
    if (!INPQ_allocate (32) ||
      !INPQ_enable_keyboard (key_down))
    {
        printf ("Unable to initialize \
                input queue manager.\n");
        exit (1);
    }

    for (;;)
    {
        if (INPQ_dequeue (&event, key_down))
        {
            if (event.data.kbd.scancode ==
            ky_ESC)
                break;
            printf ("%c\n",
                INPQ_ascii (&event));
        }
    }
    INPQ_release ();
}
```

unique binary value depending upon the event in question. The values are assigned from the `EVENT_TYPE` enumeration defined in Listing 2.

Each event type has some unique information assoicated with it. This information is stored in the event's data field. The information returned by keyboard events, for example, is the scan code of the key that was pressed or released. The information associated with some events may not be complete without additional context information. This information is stored in the attributes field. Scan codes do not take into account the state of the shift, ctrl, and alt keys. These keys must be registered by the keyboard handler and their state passed along in the attributes field of the event.

Finally, each event contains a timestamp obtained from the CMOS clock at the time the event is queued. The timestamp serves two purposes. First, it allows the input queue manager to properly order events internally (we'll discuss this in more detail in a subsequent section). Second, it allows the client program to determine the span between two or more events, à la the fighting game example mentioned previously.

Each of the event types and their corresponding data and attributes will be discussed more thoroughly when we are discussing specific event types. For the time being, let's move on to look at the data structure on which the input queue is based, the priority queue.

Priority Queue Concepts

The client program isn't necessarily going to process events as soon as they are available, so some sort of queue is required. A priority queue data structure was chosen for a number of reasons.

The priority queue structure allows events to be processed efficiently, in order or according to event type. Read "efficiently" as "in reasonable time." Pull out those data structures books and dust them off: a priority queue is a data structure that supports insertion of new prioritized elements and deletion of the element with the smallest (or largest) priority.

The priority queue structure can be implemented to require a fixed memory overhead. In other words, all memory for events and support structures can be preallocated when the input queue manager is initialized. This precludes the input queue manager allocating and freeing memory as events are processed, eliminating the risk of the input queue manager causing memory fragmentation.

Table 1. Input Queue Manager API

Input Queue Control Functions	
`INPQ_allocate()`	Initializes the input queue manager.
`INPQ_release()`	Cleans up after the input queue manager.
`INPQ_enable_keyboard()`	Enable keyboard events.
`INPQ_disable_keyboard()`	Disable keyboard events.
`INPQ_enable_mouse()`	Enable mouse events.
`INPQ_disable_mouse()`	Disable mouse events.
`INPQ_enable_timer()`	Enable timer events.
`INPQ_disable_timer()`	Disable timer events.
`INPQ_enable_user()`	Enable user-defined events.
`INPQ_disable_user()`	Disable user-defined events.
`INPQ_events_enabled()`	Return enable events.
Queue Management Functions	
`INPQ_dequeue()`	Dequeue an event, based upon event type.
`INPQ_enqueue()`	Enqueue an event, possibly user defined.
Keyboard Support Functions	
`INPQ_ascii()`	Translate a scan code to its ASCII equivalent.
Mouse Support Functions	
`INPQ_show_mouse()`	Show the mouse cursor.
`INPQ_hide_mouse()`	Hide the mouse cursor.
`INPQ_mouse.visible()`	Return true if the mouse is currently visible.
`INPQ_obscure_mouse()`	Hide mouse cursor without affecting events.
`INPQ_unobscure_mouse()`	Show mouse cursor without affecting events.
`INPQ_set_graphics_cursor_shape()`	Set the mouse cursor shape.
`INPQ_set_mouse_position()`	Reposition mouse to new coordinates.
`INPQ_get_mouse_position()`	Return mouse's current coordinates.
`INPQ_push_mouse_regions()`	Prepare to define hotspots.
`INPQ_define_mouse_region()`	Define a hotspot.
`INPQ_pop_mouse_regions()`	Discard mouse regions.
Timer Support Functions	
`INPQ_set_alarm()`	Enable one of 16 countdown timer alarms.

Listing 2. Excerpts from INPUTQ.H

```
#include "basic_dt.h"
#include "scancode.h"

#define LAST_ALARM 15

typedef enum {
    null_event      = 0x0000,
    key_down        = 0x0001,
    key_up          = 0x0002,
    mouse_down      = 0x0004,
    mouse_up        = 0x0008,
    mouse_move      = 0x0010,
    timer_alarm     = 0x0020,
    first_usr       = 0x0040,
    last_usr        = 0x8000
} EVENT_MASK;

/*
** Attributes Structures
*/

typedef struct {
    u16 k_extended      : 1;
    u16 k_shift         : 1;
    u16 k_ctrl          : 1;
    u16 k_alt           : 1;
    u16 k_unused        : 12;
} KEYBOARD_ATTR;

typedef struct {
    u16 m_region        : 8;
    u16 m_left          : 1;
    u16 m_right :       : 1;
    u16 m_center        : 1;
    u16 m_unused        : 5;
} MOUSE_ATTR;

typedef union {
    KEYBOARD_ATTR   kbd;
    MOUSE_ATTR      mouse;
    u16             value;
} INPQATTR;

/*
** Data Structures
*/

typedef struct {
    u8  scancode;
    u8  reserved1;
    u16 reserved2;
} KEYBOARD_DATA;

typedef struct {
    s16 x;
    s16 y;
} MOUSE_DATA;

typedef struct {
    u8  alarm;
    u8  reserved1;
    u16 reserved2;
} TIMER_DATA;

typedef union {
    KEYBOARD_DATA kbd;
    MOUSE_DATA  mouse;
    TIMER_DATA  timer;
    s32         value;
} INPQDATA;

/*
** Event structure
*/

typedef struct {
    u32 timestamp;
    u16 type;
    INPQATTR attr;
    INPQDATA data;
} EVENT;
```

Our priority queue implementation will be based upon a specific type of binary tree data structure known as a heap. A heap is a binary tree that satisfies the heap condition: the priority of any given node is less than or equal to the priorities of the node's children (if they exist). This condition can easily be encapsulated in an array representation, as shown in Figure 3.

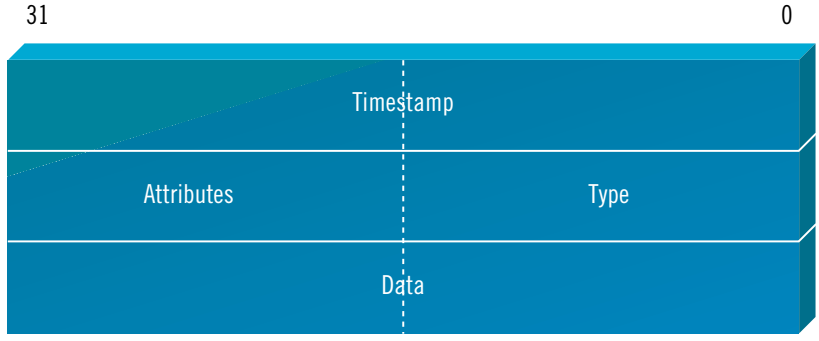
Assume that the values within the nodes are the priorities. It is trivial to verify that every node in this tree satisfies the heap condition, and therefore this tree is a heap. Now we can convert this tree representation into an array representation by simply starting at the root of the tree and adding elements to the array in a top-to-bottom, left-to-right, fashion (a breadth-first traversal if you will).

Within the array, the children for any node, say the node at position k, are located at positions 2k and 2k+1. This in turn implies that the parent of any node, say the node at position j, can be found at position `(int)`(j/2). For example, in Figure 2, the children of the node located at position 3 are located at positions 2 * 3 = 6 and 2 * 3 + 1 = 7 respectively. Conversely, the parent of the node located at position 7 is `(int)`(7/2) = 3.

Given that we can find the parent and children of any node regardless of its location on the heap, how do we perform insertion and deletion operations? Listing 3 contains the heap manager code, including code to perform insertion (enqueue) and deletion (dequeue) operations on the heap.

To insert an element into a heap, we see that the `enqueue()` function starts by inserting the new element at the end of the array. This corresponds to placing the new element at the bottom of the heap. At this point, we may not even have a heap anymore, because the heap condition may be violated at this terminal position. The routine `reorder_heap()` is called to fix any violations of the heap condition. It does this by checking the priority of the newly inserted event against the priority of its parent. If the parent priority is larger, the two events are swapped. This continues until the new event can no longer be swapped with its parent, resulting in a tree that satisfies the heap condition.

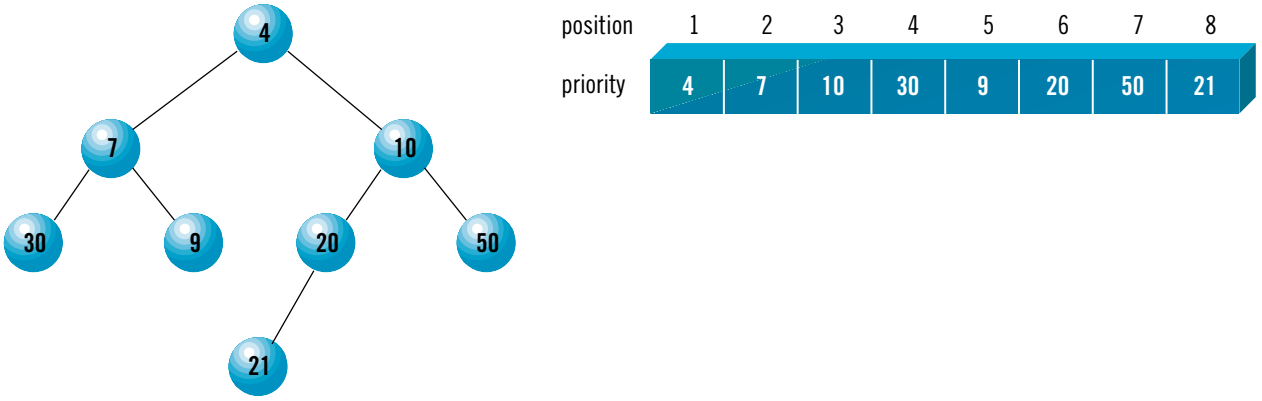
Figure 2. Event Structure



Deleting an element from the queue is just slightly more complicated. The `dequeue()` operation first deallocates the event to be deleted, returning it to the pool of available events. Then, the event at the last array index in the heap is inserted in the position of the deleted event. Again, we may no longer have a valid heap, since the event we just inserted has an arbitrary priority. The `reorder_heap()` routine is again called to restore the heap to a valid state.

This time though, `reorder_heap()` must try and work the inserted event down

Figure 3. Conversion of a Heap Represented as a Tree into an Array



the heap until the heap condition is satisfied. It does this by checking the children of the newly inserted event. If no children exist, the heap condition is satisfied, and the routine is done.

If children exist, the newly inserted event's priority is checked against the largest of its children's priorities. If the newly inserted event's priority is greater, it and the larger child are swapped. This continues until there are no more children to check (at which point the event is at the bottom level of the heap) or the event's priority is less than both of its children's priorities, satisfying the heap condition.

The final operation we want the priority queue to perform is a search for an arbitrary element. The routines `locate_event()` and `find_event()` work together to perform a recursive binary search of the heap based upon an event type. I used a recursive search because it was easy—I didn't anticipate a burgeoning heap and I was getting lazy. Needless to say, a better implementation would use a state variable and a loop to remove the dangers associated with the recursion.

Both the insertion and deletion operations can be performed in time proportional to O(ln N). The binary search operation, for an arbitrary value, can be performed in time proportional to O(2ln N) on average, with a worst case time proportional to O(N).

To provide some basis for comparison, consider implementing the priority queue with a standard, albeit modified, queue structure. Insertion would require nothing more than adding a new element to the end of the queue (remember, we want to delete items with the smallest priorities). Assuming we keep a pointer to the tail of the queue, this operation can be performed in constant time (O(1)). Deleting the element with the smallest priority is also trivial: just remove the element at the head of the list. Again, this is a constant time operation. But if you want to find an arbitrary element, searching the queue is going to take time proportional to O(N) on average.

The heap implementation takes slightly longer to insert and delete elements, because it must maintain the heap condition, but when searching for arbitrary elements, it is significantly faster on average than the standard queue implementation.

Table 2. CMOS Get Tick Count Function

```
Int 1AH Function 00H
Get Tick Count
Returns with the contents of the clock tick counter.
Call with:
AH      = 00H
Returns:
AL      = rolled-over flag
          00H if midnight not passed since last read
          <> 00H if midnight passed since last read
CX:DX   = tick count (high 16 bits in CX)
```

The Event Pool

As I mentioned previously, `INPQ_allocate()` and `INPQ_release()`, shown in Listing 4, work together to allocate and free memory for events and the input queue. Once allocated, events are stored in a pool, which is managed by the `allocate_event()` and `deallocate_event()` routines.

`Allocate_event()` is called from our event handlers when input is received. If there is an event to return, the CMOS timer function, `GetTickCount` (also shown in Table 2) is consulted and its current value is stored in the timestamp field of the event.

The timestamp value is important in two ways. First and foremost, it is used to order the input queue. The queue is ordered from smallest timestamp (oldest event) to largest timestamp (newest event). All queued events have nonzero timestamp values, so this field is also used as a free indicator in the event pool. Any event with a zero timestamp cannot possibly reside on the input queue and is therefore available for allocation.

Before it returns the address of the new event to its caller, `allocate_event()` enqueues the event. The only responsi-

bility the caller has is to fill in the event type, data, and attributes fields. `Deallocate_event()` does little more than reset the timestamp of the supplied event to zero, freeing it for later use.

Taking Control of the Keyboard

Now that we can allocate and deallocate events, we're ready to start talking about populating the queue. The input queue manager has three event handlers—one each for the keyboard, the system timer, and the mouse. The event handlers for the keyboard and the system timer both grab interrupt vectors and install interrupt service routines to gain control of all input from the given device. The mouse handler is slightly different. It makes use of the existing mouse device driver for its event notification.

The final topic we're going to discuss is the keyboard event handler. We'll cover the timer and mouse event handlers in the second part of this series.

The IBM PC keyboard communicates with the PC BIOS via hardware interrupt `09h`. Whenever a key is pressed or released, an interrupt `09h` is generated and in most cases, the scan code of the key pressed can be read from the keyboard port located at address `60h`.

Certain keys generate extended scan codes. A key that generates an extended scan code requires the keyboard handler to processes two or more interrupts for the single key. On the first interrupt, the extended scan code identifier, `0E0h`, is read from the keyboard port. The actual scan code may be read from the keyboard port on the next interrupt.

In most cases, extended scan codes are used when there are multiple identical keys on the keyboard. For example, the ctrl key on the lefthand side of the keyboard returns a standard scan code, and the one on the right returns the same scan code value, but as an extended scan code combination.

Some keys, such as print screen/system request and pause/break, generate multiple extended scan code sequences. The simple keyboard handler we're developing will not look for these, though this will not really be a problem. Each subsequence is an extended scan code sequence; a single keystroke generates multiple events.

Our keyboard handler will perform the following functions:

- Obtain the scan code for the keystroke (taking into account the extended scan code behavior)
- Acknowledge to the hardware that the interrupt was received and handled
- Track the current state of the shift, ctrl, and alt keys (the keyboard handler does not differentiate between left and right instances of these keys)
- If appropriate, allocate an event and fill in the current attributes and scan code values.

Keyboard Events

Figure 4 is a graphical representation of a keyboard event. The event type field will either have bit 1 or bit 2 set to indicate a `key_down` or `key_up` event, respectively. The attributes field contains bit flags indicating whether the event was obtained as a result of an extended scan code as well as the state of the shift, ctrl, and alt keys at the time that the event was generated. Finally, the data field contains the actual scan code received.

You might ask why the ASCII code for the key isn't stored in the event as well. Originally, it was, and the event handler was spending most of its time doing the scan code to ASCII value translation. Because of the overhead and because each key is uniquely identified by its scan code, I made an executive decision to perform the ASCII translation outside the interrupt and only at the request of the client program. This method offers one more advantage: if an ASCII value is never requested (via a call to `INPQ_ascii()`), the code to perform the translation (and the substantial translation tables) never get linked into the client program's executable.

Listing 4. Excerpts from INPUTQ.C

```
BOOLEAN INPQ_allocate (s16 n)
{
    if (n <= 0)
        n = DEFAULT_QUEUE_DEPTH;

    pool = (EVENT *) calloc (n, sizeof (EVENT));
    if (!pool)
        return False;
    pool_last = n - 1;
    pool_first_free = 0;

    queue = (EVENT **) calloc (n + 1, sizeof (EVENT **));
    if (!queue)
    {
        free (pool);
        pool = NULL;
        return False;
    }
    queue_max   = n;
    queue_last  = 0;

    atexit (INPQ_release);
    return True;
}

void INPQ_release (void)
{
    INPQ_disable_keyboard (events_enabled);
    INPQ_disable_mouse (events_enabled);
    INPQ_disable_timer (events_enabled);

    if (queue)
    {
        free (queue);
        queue = NULL;
    }
    if (pool)
    {
        free (pool);
        pool = NULL
    }
}
BOOLEAN INPQ_enable_keyboard (EVENT_MASK keyboard_events)
{
    keyboard_events &= (key_down | key_up);
    if (keyboard_events)
    {
        initialize_keyboard ();
        events_enabled |= keyboard_events;
    }
    return True;
}

void INPQ_disable_keyboard (EVENT_MASK keyboard_events)
{
    keyboard_events &= (key_down | key_up);
    events_enabled &= ~keyboard_events;
    if (!(events_enabled & (key_down | key_up)))
        release_keyboard ();
}
```

The Keyboard Event Handler

Listing 5 contains the code for the keyboard event handler. The first two routines are used to initialize and release the keyboard interrupt, respectively. The `keyboard_events()` routine, as the name implies, gathers keyboard events and returns them on the input queue.

The first thing that `keyboard_events()` does is grab the byte from the keyboard port and clear the interrupt controller:

```
scancode = inp (KEYBOARD_PORT);
outp (PIC_REGISTER,
  NONSPECIFIC_EOI);
        // acknowledge_interrupt
```

The interrupt controller must be cleared because interrupt `09h` is a hardware interrupt. It is actually attached to a line on the programmable interrupt controller (PIC). Whenever the PIC generates an interrupt, that interrupt is masked off until the handler informs it that the interrupt has been noticed. The interrupt is acknowledged by outputting a value of `0x20` (nonspecific EOI) to the PIC control port at address `0x20`.

Originally, this operation was performed at the end of the interrupt handler, which I consider a much safer place for it. But as soon as I had the debugger stop on a breakpoint within the keyboard handler, the machine would hang, and no further keyboard input was possible. This was because the PIC refused to allow any more keyboard interrupts to occur until it saw an EOI for the current interrupt. But I couldn't make the debugger continue to the point of the EOI until I could use the keyboard again (a frustrating catch-22).

After the interrupt has been acknowledged, the scan code is checked against the extended scan code. If it matches, we record the fact and exit the handler, knowing that the next interrupt we receive will contain the actual scan code.

If the scan code isn't the extended scan code, the handler determines the event type (`key_down` or `key_up`) and the state of the special keys (shift, ctrl, and alt). This is all the information we need to generate an event. But we only generate an event if the current event type matches one of the currently enabled event types.

If the input queue manager is enabled for the event that's been received, an event structure is allocated with a call to `allocate_event()`, and the keyboard attributes and data fields are filled in. If the input queue is full (all events have been allocated, but the user hasn't dequeued any at the time we want to post an event), we increment the `lost_input` variable. This variable isn't currently looked at anywhere, but I figured that eventually, it might be a useful debugging aid.

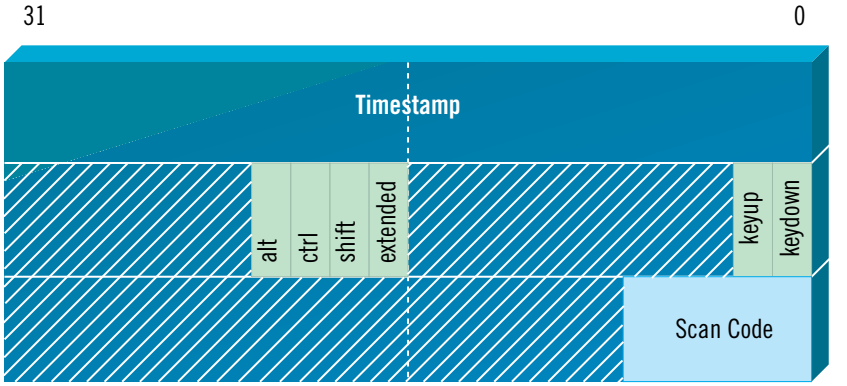
Where We're Going

That does it for now. We've covered the basic input queue manager, the event structures, and how keyboards events are actually generated. Next time, we'll discuss user-defined events, timer events,

Listing 5. KBD.C

```
static void interrupt (*oldint9h)(void) = NULL;
static u16 static_attributes = 0;

BOOLEAN initialize_keyboard (void)
{
    if (!oldint9h)
    {
        oldint9h = getvect (KBD_INT);
        setvect (KBD_INT, keyboard_events);
        return True;
    }
    return False;
}

void release_keyboard (void)
{
    if (oldint9h)
    {
        setvect (KBD_INT, oldint9h);
        oldint9h = NULL;
    }
}

static void interrupt keyboard_events (void)
{
    register u8 scancode;
    u16 evt_mask = 0;

    scancode = inp (KEYBOARD_PORT);
    //  acknowledge_interrupt
    outp (PIC_REGISTER, NONSPECIFIC_EOI);

    if (scancode == EXTENDED_SCAN_CODE)
    {
        static_attributes |= MOD_EXTENDED;
        return;
    }

    // assume key down event, modify if we
    // find out otherwise.
    evt_mask |= key_down;
    if (scancode & 0x80)
        evt_mask <<= 1;

    // clear key up flag in scancode
    scancode &= 0x7F;

    if (evt_mask & key_down)
    {
        // key press modifiers
        switch (scancode) {
            case ky_CTRL              :
                static_attributes |= MOD_CTRL;
                break;
            case ky_ALT               :
                static_attributes |= MOD_ALT;
                break;
            case ky_LEFT_SHIFT        :
            case ky_RIGHT_SHIFT       :
                static_attributes |=
MOD_SHIFT;
                break;
            default                   :
                break;
        }
    }
    else
    {
        // key release modifiers
        switch (scancode) {
            case ky_CTRL              :
                static_attributes &=
~MOD_CTRL;
                break;
            case ky_ALT               :
                static_attributes &= ~MOD_ALT;
                break;
            case ky_LEFT_SHIFT        :
            case ky_RIGHT_SHIFT       :
                static_attributes &=
~MOD_SHIFT;
                break;
            default                   :
                break;
        }
    }

    if (events_enabled & evt_mask)
    {
        EVENT *event = allocate_event ();

        if (event)
        {
            event->event_mask = evt_mask;
            event->data.value = scancode;
            event->attributes.value =
                static_attributes;
        }
        else
            ++lost_input;
    }

    static_attributes &= ~MOD_EXTENDED;
}
```

Figure 4. Keyboard Event Structure



and mouse events. We'll also go into the mouse cursor interface, which will allow us to self-draw the mouse cursor in any video mode.

Because of space limitations inherent in a magazine format, there is no way that the entire source for the input queue manager could be included with this article. Further, most of the comments were stripped out of the source that is included. Therefore, I encourage you to download the full source from the *Game Developer* ftp site (ftp://ftp.mfi.com/pub/gamedev/src) or on CompuServe SDFORUM's *Game Developer* library. If you don't have access to either of these resources, but you do have an e-mail address, I'd be happy to e-mail a uuencoded version of the sources to you. Just send mail to inpq-source@irvine.com with a subject line of "inpq source." ■

Mike Michaels is a senior software engineer with Irvine Compiler Corp., a small company that produces ADA compilers. Yes, he realizes that Ada is dead. No, he doesn't write games in Ada. You can contact him via e-mail at mike@irvine.com.

REFERENCES

Algorithms, Second Edition; **Robert Sedgewick, Addison Wesley, 1988**
This book provided a wealth of information when I was trying to remember how priority queues worked—that data structures class was a long, long time ago!

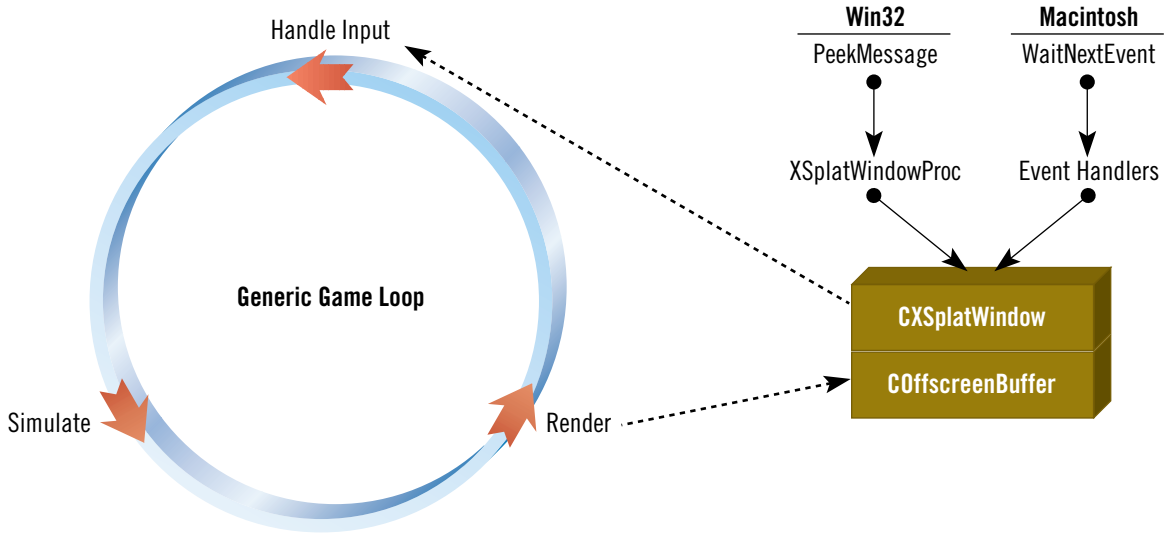
PC Game Programmers Encyclopedia v. 1.0a; **Mark Feldman, et. al. 1994**
The PCGPE—is a collection of text files and a viewer program—can be located at ftp://x2ftp.oulu.fi:/pub/msdos/programming/gpe/. The viewer and some of the text files were written by Feldman. Other files were garnered from sources throughout the Internet. Topics range from basic tutorials on assembly language to algorithmic explanations of texture mapping. I found this to be an indespensible guide when it came time to write the keyboard and timer interrupt handlers.

Advanced MS-DOS Programming, Second Edition; **Ray Duncan, Microsoft Press, 1988**
This volume's reference section lists all the BIOS and MS-DOS interrupt functions you are likely to need. While the text glosses over mouse programming, the reference guide lists all the mouse driver functions.

Microsoft Macro Assembler Reference v. 6.0; **Microsoft Corp.**
I used this volume to create the scan code to ASCII translation tables used by the `INPQ_ascii()` routine.

XSplat: A Foundation for Cross-Platform Development

Figure 1. The XSplat generic game architecture



The UC Theatre in Berkeley, Calif. plays kung fu double features every Thursday. Recently, I had the pleasure of catching *Twin Dragons*, in which Jackie Chan plays twin brothers accidentally separated at birth in Hong Kong. One moves to the U.S. with his wealthy parents, studies at posh New York private schools, and grows up to be an acclaimed piano player with very refined tastes. The other is found by a prostitute and grows up on the streets of Hong Kong to become an expert fighter. Neither brother knows of the other's existence.

Eventually, of course, the brothers meet. They become fast friends and learn to take advantage of their identical appearances for maximum comic effect. The kung fu master receives a standing ovation for his passionate performance as conductor of a major orchestra, and the musician brother helps to defeat the forces of a major organized crime boss. All the while, Jackie Chan pulls off plenty of the insane gymnastic stunts that make his movies so painfully enjoyable.

The "twins separated at birth" plot is awfully predictable. Every few years we see a new interpretation. I'm quite sure you've caught a few choice scenes from the 90s courtroom remake, *Apple vs. Microsoft*.

Separated at Birth

I would gladly bet that for the last several years, Microsoft and Apple have employed more lawyers, clerks, judges, witnesses, paralegals, stenographers, and pizza makers while arguing over their claims and counterclaims than there are

middle managers at IBM. The Battle of the Operating Systems rages between the twin children of Xerox PARC as they struggle over the look and feel of modern GUI computing.

As the distance between Apple and Microsoft shrinks, each begins to exert a strange influence on the other. Windows 95 introduces an integrated desktop metaphor, complete with a drag-and-drop recycling bin, plug-and-play, long filenames, and shortcuts to programs and documents. In Copland (MacOS 96?), Apple promises a true kernel-based operating system with a device driver layer, protected memory, and user interface elements such as the ability to minimize windows or gather them in "trays" at the bottom of the screen.

All this belies the secret truth it's taken hundreds of lawyers to obscure: Windows and the Macintosh operating system are identical twins, born of the same womb and separated at birth. QuickTime, meet Video for Windows. QuickDraw, GDI. OpenDoc, OLE. Component, DLL. PPC, DDE. What's the ultimate difference? As programmers, we can create a single lollipop that will make both twin babies happy.

I'm being naive on purpose to make a point: games generally bypass a large portion of the operating system in favor of algorithms specific to their application. Show me a system capable of displaying 256-color palletized images and handling user input via a mouse and keyboard, and I'll show you an ideal platform for today's games and most of tomorrow's. Tack on a rich set of standard system services such as event management, a 32-bit flat memory model, a heap manager, and a file I/O system, throw in a 98% combined market share, and almost any game programmer will be in heaven.

In this five-part series, I'll explore and develop a simple, reusable, platform-independent, GUI-based game library for the Macintosh and Windows that gives us something close to this ideal system. For historical reasons, I've chosen to call this small library XSplat.

This is actually the second article of the series, which I began with "A C++ Class for Cross-Platform Double Buffered Graphics" (June/July 1995). Subsequent articles will continue to build on each other until we have created a full-fledged playable game whose core code has been written for XSplat and compiles for Macintosh and Windows with no additional effort. The game won't be a cutting-edge mega-technological hit, but it will be a full-featured one that (I have high hopes) will be fun and playable.

Developing a complete cross-platform game library involves a lot of code. Generally, there will be more code associated with these articles than we can print here, but you'll find complete source code and compiled executables on the *Game Developer* ftp site, ftp://ftp.mfi.com/pub/gamedev/src/, in the /xsplat directory. All code will compile on the Macintosh using Metrowerks C/C++ and on Windows using Microsoft Visual C++ v. 2.1.

The System Speaks

The most significant feature of any standard Windows/Mac application is a loop that waits for the operating system to percolate events up from the user. When

Jon Blossom

With operating systems becoming increasingly similar, complete cross-platform development is essential. In the first of a five-part series, Jon Blossom lays the groundwork with a library called XSPLAT.

Listing 1. XSplat Windowing and Messaging System Declaration

```
class CXSplatWindow
{
public:
   CXSplatWindow(char const *Caption, int WindowWidth=0,
      int WindowHeight=0, unsigned char const *Colors = 0);
   ~CXSplatWindow(void);

   void Activate(void);
   void Deactivate(void);
   void Terminate(void);

   void KeyDown(char Key, int RepeatCount);

   void MouseDown(int x, int y);
   void MouseUp(int x, int y);
   void MouseMove(int x, int y);

   COffscreenBuffer *GetOffscreenBuffer(void)
      { return pOffscreenBuffer; };

protected:
   int IsActiveFlag;
   int MouseDownFlag;
   COffscreenBuffer *pOffscreenBuffer;

#if defined(_WINDOWS)
// Windows implementation details

   HPALETTE Palette;
   HWND Window;

   friend LONG PASCAL XSplatWindowProc(HWND, UINT, WPARAM, LPARAM);

#elif defined(_MACINTOSH)
// Macintosh implementation details

   WindowPtr Window;
   PaletteHandle WinPalette;

   friend void HandleMouseDown(EventRecord *pEvent);

#endif
};

// This is the XSplat message processing system
void DispatchEvents(void);

// This is the XSplat main function, called from WinMain or
// from main after platform-specific setup
void XSplatMain(void);
```

the loop sees a message it cares about, it calls out to some other set of functions to process the event, then returns its ear to the queue and continues until the user exits the application. Pretty basic stuff. All we have to do is plug in to the appropriate queue.

On both systems, windows provide the atomic units of graphical applications, so most messages are tied to individual windows as the user performs actions on them. The application just creates a window, calls on the system to process events, handles game logic, and renders until the program terminates. Our goal today is to encapsulate all this in a platform-independent game loop that looks like this:

```
void XSplatMain(void)
{
  // Create a 320x240 window with the
      title "XSplat Window"
   CXSplatWindow *pWindow = new
     CXSplatWindow("XSplatWindow", 320,
     240);

   if (pWindow)
   {
    while (GameIsRunning)
    {
      DispatchEvents();
      DoGameLogic();
      Render();
    }

    // Get rid of the window for good
    delete pWindow;
   }
}
```

All the logic of the user interface takes place in the `CXSplatWindow` object. `DispatchEvents` translates messages from the system into `CXSplatWindow` methods, `DoGameLogic` controls the actual flow of the non-event-based portions of the game, if any, and `Render` displays the current game state on the screen. Figure 1 shows this platform-independent architecture.

Event notification differs significantly between the Macintosh and Windows systems. The Macintosh pours all its messages through one pipe, requiring that the application filter and distribute messages at the receiving end of the pipe. In Windows, the message filter sits at the sending end of the pipe, and the system eases the burden by passing messages out to individual windows instead of to the controlling application.

Big deal. `CXSplatWindow` provides an adapter that plugs onto either end and redirects messages however we want them. This funnel adapter can even handle many system messages itself, passing

Listing 2. CXSplatWindow Constructor for Win32

```
// This will be used as the window class name for CXSplatWindows
static char XSplatWindowClassName[] = "XSPLAT";

// This will be the window style of all new created CXSplatWindows
// For now, don´t allow the window to be resized
static const DWORD XSplatWindowStyle =
    WS_OVERLAPPEDWINDOW & ~WS_THICKFRAME;

CXSplatWindow::CXSplatWindow(const char *Caption, int WindowWidth,
    int WindowHeight, unsigned char const *Colors) :
    Palette(0), Window(0), pOffscreenBuffer(0),
    IsActiveFlag(0), MouseDownFlag(0)
{
    extern HINSTANCE ghInstance;
    assert(ghInstance);

    // Register the XSplatWindow class if it isn´t already registered
    int Success = 1;
    WNDCLASS ClassInfo;
    if (!GetClassInfo(ghInstance, XSplatWindowClassName, &ClassInfo))
    {
        ClassInfo.hCursor          = LoadCursor(0, IDC_ARROW);
        ClassInfo.hIcon            = LoadIcon(ghInstance, IDI_APPLICATION);
        ClassInfo.lpszMenuName     = 0;
        ClassInfo.lpszClassName    = XSplatWindowClassName;
        ClassInfo.hbrBackground    = HBRUSH(GetStockObject(WHITE_BRUSH));
        ClassInfo.hInstance        = ghInstance;
        ClassInfo.style            = CS_VREDRAW | CS_HREDRAW | CS_OWNDC;
        ClassInfo.lpfnWndProc      = WNDPROC(XSplatWindowProc);
        ClassInfo.cbWndExtra       = 0;
        ClassInfo.cbClsExtra       = 0;

        Success = RegisterClass(&ClassInfo);
    }

    if (Success)
    {
        // Determine screen dimensions
        int ScreenWidth = GetSystemMetrics(SM_CXSCREEN);
        int ScreenHeight = GetSystemMetrics(SM_CYSCREEN);

        // Given a zero or negative dimension, fill the screen
        if (WindowWidth <= 0)
            WindowWidth = ScreenWidth;
        if (WindowHeight <= 0)
            WindowHeight = ScreenHeight;

        // Determine the window size for the requested client area
        RECT WindowRect = { 0, 0, WindowWidth, WindowHeight };
        AdjustWindowRect(&WindowRect, XSplatWindowStyle, 0);

        // Make sure it´s not larger than the screen
        int WindowWidth = WindowRect.right - WindowRect.left;
        if (WindowWidth > ScreenWidth) WindowWidth = ScreenWidth;

        int WindowHeight = WindowRect.bottom - WindowRect.top;
        if (WindowHeight > ScreenHeight) WindowHeight = ScreenHeight;

        // Create the window centered on the screen
        int Left = (ScreenWidth - WindowWidth) / 2;
        int Top = (ScreenHeight - WindowHeight) / 2;

        Window = CreateWindow(XSplatWindowClassName, Caption,
            XSplatWindowStyle, Left, Top, WindowWidth, WindowHeight,
            0, 0, ghInstance, 0);

        if (Window)
        {
            // Store a pointer back to this object in GWL_USERDATA
            SetWindowLong(Window, GWL_USERDATA, (long)this);

            // TODO: Set up the palette from the given Colors
            // TODO: We´ll do this in a later article...
            // For now, create a gray wash
            struct
            {
                WORD Version;
                WORD NumberOfEntries;
                PALETTEENTRY aEntries[256];
            } LogPalette =
            {
                0x300, // Palette version
                256    // Number of colors
            };

            for (int i=0; i<256; ++i)
            {
                LogPalette.aEntries[i].peRed =
                    LogPalette.aEntries[i].peGreen =
                    LogPalette.aEntries[i].peBlue = i;

                LogPalette.aEntries[i].peFlags = 0;
            }
            Palette = CreatePalette((LOGPALETTE *)&LogPalette);

            // Initialize the Window´s DC as necessary
            // Since it´s a CS_OWNDC, these settings will last
            // forever
            HDC hdc = GetDC(Window);
            if (hdc)
            {
                SetMapMode(hdc, MM_TEXT);
                SelectPalette(hdc, Palette, FALSE);
                RealizePalette(hdc);

                // This isn´t really necessary, but...
                ReleaseDC(Window, hdc);
            }

            // All ready to go, show the window
            ShowWindow(Window, SW_NORMAL);

            // Set up the offscreen environment. Note that because
            // of the way COffscreenBuffer was originally defined,
            // the window  must be visible and active
            assert(GetActiveWindow() == Window);
            pOffscreenBuffer = new COffscreenBuffer;
        }
    }
}
```

along only a handful of important events such as keyboard and mouse input.

Listing 1 shows a declaration of a minimal `CXSplatWindow` interface, whose member functions you may want to virtualize. I've combined Macintosh and 32-bit Windows declarations using the preprocessor symbols `_WINDOWS` and `_MACINTOSH`, defined elsewhere. There's also a friend called `XSplatWindowProc` required by the Win32 `CXSplatWindow` implementation, for reasons I'll describe shortly.

Constructing Windows

A `CXSplatWindow` is more than just a message filter. It's the atomic unit of the

XSplat interface, the atomic unit of either target system. It's a visible, interactive piece of the user interface.

The `CXSplatWindow` constructor creates a window using platform-specific system functions. We pass along a title, an optional height and width, and an optional array of 768 bytes describing 256 RGB triplets (XSplat assumes an 8-bit palletized display, and we'll deal with palettes in a future article). If height or width is omitted, zero, or negative, the constructor creates a window that fills the screen in the omitted dimension. No value or a null pointer for `ColorArray` tells the constructor to use a default gray wash for the window.

So the command:

```
CXSplatWindow *pWindow = new CXSplatWindow("Testing");
```

will create a full-screen window with a gray wash and the title "Testing."

We'll want the window to display an image of our own creation, so we'll create a `COffscreenBuffer` for every `CXSplatWindow` using the code introduced in my last article (to access this code, go to the *Game Developer* ftp site at ftp://ftp.mfi.com/gdmag/src.).

The `CXSplatWindow` constructors are very simple and hide the nuances of each platform from XSplat applications. Listing 2 shows the 32-bit Windows version, and Listing 3 shows the Macintosh version. Let's take a look at what's going on in each.

In Windows, every window requires an associated window class, and every window class requires an associated window procedure. Before creating a window, the constructor makes sure the window class `XSPLAT` has been registered for a window based on the `XSplatWindowProc` procedure. This requires an instance handle, provided through a global `ghInstance` handle that we set up in `WinMain`. We'll take a look at `XSplatWindowProc`, a target for the Windows messaging mechanism, a little later.

Listing 3. CXSplat Window Constructor for Macintosh

```
CXSplatWindow::CXSplatWindow(const char *Caption,
int WindowWidth,
    int WindowHeight, unsigned char const *Colors) :
    WinPalette(0), Window(0), pOffscreenBuffer(0),
    IsActiveFlag(0), MouseDownFlag(0)
{
    // Determine the screen size
    int ScreenWidth = qd.screenBits.bounds.right -
        qd.screenBits.bounds.left;
    int ScreenHeight = qd.screenBits.bounds.bottom -
        qd.screenBits.bounds.top;

    // Given a zero or negative dimension, fill the
    // screen
    if (WindowWidth <= 0)
        WindowWidth = ScreenWidth;
    if (WindowHeight <= 0)
        WindowHeight = ScreenHeight;

    // Create the window centered on the screen
    int Left = (ScreenWidth - WindowWidth) / 2;
    int Top = (ScreenHeight - WindowHeight) / 2;

    Rect WindowRect = {Top, Left,
        Top + WindowHeight, Left + WindowWidth};

    // Convert the caption to a pascal string
    // TODO: This doesn't do well with non-ASCII
    // character sets...
    char unsigned PascalCaption[256];
    strcpy((char *)&PascalCaption[1], Caption);
    PascalCaption[0] = (char unsigned)strlen(Cap-
tion);

    // Create a new color document window with these
    // dimensions
    Window = NewCWindow(0, &WindowRect, PascalCap
      tion, TRUE, noGrowDocProc, WindowPtr(-1),
      TRUE, 0);
    if (Window)
    {
        // Store a pointer back to the window
        SetWRefCon(Window, (long)this);

        // TODO: Set up the palette from the given
        // Colors
        // TODO: We'll do this in a later article...
        // For now, create a gray wash
        // Create a palette for the window and make
        // sure it will give a 1:1 color mapping
        // with pmExplicit | pmAnimated.
        WinPalette = NewPalette(256, 0, pmExplicit |
pmAnimated, 0);
        if (WinPalette)
        {
            for (int i=0; i<256; ++i)
            {
                RGBColor rgb;
                rgb.red = i << 8;
                rgb.green = rgb.red;
                rgb.blue = rgb.red;

                SetEntryColor(WinPalette, i, &rgb);
            }
            SetEntryUsage(WinPalette, 0, pmExplicit
| pmAnimated, 0);
            SetEntryUsage(WinPalette, 255, pmExplic-
it | pmAnimated, 0);
            SetPalette(Window, WinPalette, FALSE);
        }

        // Initialize the Window
        SetPort(Window);
        ForeColor(blackColor);
        BackColor(whiteColor);
        PenNormal();

        // Set up the offscreen
        // environment. Note that because
        // of the way COffscreenBuffer was
        // originally defined, the window
        // must be visible and active
        pOffscreenBuffer = new COffscreen-
Buffer;
    }
}
```

The `XSPLAT` window class uses the `CS_OWNDC` style; changes to the window's device context will remain across `GetDC/ReleaseDC` calls. On Windows 3.1 (and on Win32s), `CS_OWNDC` windows eat up precious system resources, but it's well worth it for this type of application. On Windows NT and Windows 95, there's not really anything to worry about.

Once it has a window to work with, the Windows constructor creates a gray wash palette using a fake `LOGPALETTE` structure on the stack, initializes the window's device context, and creates an associated `COffscreenBuffer`. When the window is ready, it's revealed to the user.

The Macintosh `CXSplatWindow` constructor does almost exactly the same thing. It creates a window with the given dimensions using the system call `NewCWindow`, creates a gray wash palette associated with the window, initializes the drawing system for the window, and attaches a `COffscreenBuffer` object.

The `CXSplatWindow` object holds a system-specific reference to the associated system-specific window (`HWND` on Windows and `WindowPtr` on Macintosh). However, the message dispatching functions need to associate a `CXSplatWindow` object with a system-specific window. To do this, we'll take advantage of functions on both platforms that attach 32 bits of application-specific data to any window. On Windows, the API to do this is `SetWindowLong`. Its Macintosh twin is `SetWRefCon`. XSplat uses these 32 bits to

store a pointer back to a CXSplatWindow object, used by DispatchEvents to reroute system-specific messages to more general CXSplatWindow controllers.

Two other member variables, IsActiveFlag and MouseDownFlag need some examination. These variables (I've made them individual integers for simplicity) will store information received as a result of MouseDown/MouseUp and Activate/Deactivate events. There's a lot more to cover before we can address those functions, though. Specifically, we need a message switchboard to translate system events into calls to CXSplatWindow objects.

Passing Messages

The XSplat function DispatchEvents handles the polling and processing of system messages. To build a single-window application, just create a CXSplatWindow object, and DispatchEvents will do the rest.

On Windows, DispatchEvents performs a standard PeekMessage/TranslateMessage/DispatchMessage dance. The real event filtering occurs in the XSplatWindowProc procedure, the recipient of all messages dispatched to XSPLAT windows this way. When a message of interest enters XSplatWindowProc, the procedure passes it on to the CXSplatWindow object associated with the target window. Unless some evil application has mucked with the window's instance data, we can be sure of finding a valid pointer to the CXSplatWindow object using GetWindowLong.

On the Macintosh, DispatchEvents calls WaitNextEvent to look for events. As long as it finds events in the queue, it either handles these according to some default behavior or passes them on to the appropriate CXSplatWindow, retrieved through a GetWRefCon call. This is exactly the same process as DispatchEvents under Windows, without the XSplatWindowProc step in the middle.

The Macintosh XSplat switchboard has to be smarter than its Windows counterpart because the operating system doesn't always associate events with windows. In particular, there's no real support for the promised mouse movement events. WaitNextEvent can tell you when the mouse leaves a given region, but we want to know whenever and wherever the

Listing 4. Some of the Win Dispatch Architecture (Con't. on p. 36)

```
// Windows DispatchEvents is very minimal. It just forces the
// system to pass events on to XSplatWindowProc
void DispatchEvents(void)
{
    MSG Message;

    while (PeekMessage(&Message, 0, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&Message);
        DispatchMessage(&Message);
    }
}
LONG PASCAL XSplatWindowProc(HWND Window, UINT Message, WPARAM wParam,
    LPARAM lParam)
{
    // You don´t really need the pXSplatWindow to process all
    // messages -- most just go through DefWindowProc anyway. If you´re
    // concerned, move this into the messages that really care.
    CXSplatWindow *pXSplatWindow =
        (CXSplatWindow *)GetWindowLong(Window, GWL_USERDATA);

    switch(Message)
    {
        case WM_ACTIVATE:
            // Re-realize the palette on activate
            if (pXSplatWindow)
            {
                HDC hdc = GetDC(Window);
                SelectPalette(hdc, pXSplatWindow->Palette, FALSE);
                ReleaseDC(Window, hdc);

                // Let the CXSplatWindow know that it´s becoming active
                if (LOWORD(wParam) != WA_INACTIVE)
                    pXSplatWindow->Activate();
                else
                    pXSplatWindow->Deactivate();
            }
            break;

        case WM_CHAR:
            pXSplatWindow->KeyDown((char)wParam, LOWORD(lParam));
            break;

        case WM_CLOSE:
        case WM_DESTROY:
            if (pXSplatWindow)
            {
                // Avoid a bad delete/WM_DESTROY loop
                // Reset the pointer back here to avoid confusion
                pXSplatWindow->Window = 0;
            }
            GameRunningFlag = 0;
            break;

        case WM_LBUTTONDOWN:
            SetCapture(Window);
            if (pXSplatWindow)
                pXSplatWindow->MouseDown(LOWORD(lParam), HIWORD(lParam));
            break;

        case WM_LBUTTONUP:
            ReleaseCapture();
            if (pXSplatWindow)
                pXSplatWindow->MouseUp(LOWORD(lParam), HIWORD(lParam));
            break;

        case WM_MOUSEMOVE:
            if (pXSplatWindow)
                pXSplatWindow->MouseMove(LOWORD(lParam), HIWORD(lParam));
```

Listing 4. Some of the Win Dispatch Architecture (Con't. from p. 35)

```
            break;

        case WM_PAINT:
            // Since we've always got a complete picture of the game around
            // in the offscreen buffer, default paint handling poses no
            // problem: copy the offscreen buffer, if any, to the screen

            PAINTSTRUCT Paint;
            HDC hdc;
            hdc = BeginPaint(Window, &Paint);

            if (pXSplatWindow)
            {
                COffscreenBuffer *pOffscreen =
                    pXSplatWindow->pOffscreenBuffer;
                if (pOffscreen)
                {
                    // Lock and Unlock are no-ops on Windows, but include
                    // them for good measure.
                    pOffscreen->Lock();
                    pOffscreen->SwapBuffer();
                    pOffscreen->Unlock();
                }
            }

            EndPaint(Window, &Paint);
            break;

        case WM_PALETTECHANGED:
            // Ignore palette changed messages for this window
            if ((HWND)wParam == Window)
                break;

            // For others, re-realize palette
        case WM_QUERYNEWPALETTE:
            if (pXSplatWindow)
            {
                HDC hdc = GetDC(Window);

                SelectPalette(hdc, pXSplatWindow->Palette, FALSE);
                BOOL Redraw = RealizePalette(hdc);
                ReleaseDC(Window, hdc);

                if (Redraw)
                    InvalidateRect(Window, 0, FALSE);

                return Redraw;
            }
            break;
    }

    return DefWindowProc(Window, Message, wParam, lParam);
}
```

mouse changes position anywhere on the screen, relative to the origin of the window's content region. So every time it goes through the loop, `DispatchEvents` checks the current mouse position and passes it through to the appropriate `CXSplatWindow` if it has changed.

But how do we decide on an "appropriate window?" In Windows, we can capture the mouse using `SetCapture` after a mouse down event so that all subsequent mouse events go to that window, then release the capture when the button comes up. Because the Macintosh version generates its own mouse events, it also must keep its own capture information.

When a mouse down event occurs in the content region of a window, `HandleMouseDown` stores a pointer to the `CXSplatWindow` receiving the event. `DispatchEvents` passes subsequent mouse events to that `CXSplatWindow`, resetting the capture pointer on the next mouse up event. If there's no window receiving the capture, `DispatchEvents` will pass the events on to the `CXSplatWindow` associated with the front window.

On both platforms, `DispatchEvents` provides default behaviors for application activation and deactivation, window dragging and resizing, and basic system events. It also handles window updates by copying the `COffscreenBuffer` to the screen through standard methods. All the application has to deal with are the aspects that make it unique: responding to the mouse, pausing the game on deactivation, and constructing the off-screen image.

Unfortunately, there's not enough space here to print or describe the two systems in their entirety. To give you the general feel for what's going on, Listings 3 and 4 show important pieces of the XSplat event management functions for both platforms. Events coming through system-specific channels such as `WM_LBUTTONDOWN` or `mouseDown` notifications are turned into calls to `CXSplatWindow` objects, which don't have to worry about the source of the message. Again, complete source code is available on ftp.mfi.com.

A Complete Application

Let's create a complete application using the XSplat system so far, just to prove that it can actually be done. Since all the messaging, window creation, and double buffering has been set up, we only have to implement `XSplatMain`, Windows `WinMain`, and `Macintosh main`, then we need to implement a `CXSplatWindow` object.

We need a simple application that's going to use the architecture so far. How about a rudimentary paint program? This application will present a small (320-by-240) window filled with the colors of the palette (a gray wash, for now). As you click the mouse button and drag the cursor across the window, the application will invert the pixels it touches, and you'll be able to draw simple pictures.

First, we'll need an `XSplatMain` that sets up a `CXSplatWindow` and fills it with a cycle of palette colors before entering a standard message processing loop.

Listing 5. A Very Simple Drawing Application Using XSplat

```
// GameRunningFlag is reset by the event handling system when
// the user quits the game.
int GameRunningFlag = 1;

void XSplatMain(void)
{
    // Create a 320x240 window with no specific colors
    CXSplatWindow *pWindow =
        new CXSplatWindow("XSplat Test Application", 320, 240);

    if (pWindow)
    {
        // Fill the buffer with the palette in horizontal lines
        COffscreenBuffer *pBuffer = pWindow->GetOffscreenBuffer();
        if (pBuffer)
        {
            pBuffer->Lock();
            char unsigned *pBits = pBuffer->GetBits();
            for (int y=0; y<pBuffer->GetHeight(); ++y)
            {
                memset((void *)pBits, y % 256, pBuffer->GetWidth());
                pBits += pBuffer->GetStride();
            }
            pBuffer->Unlock();
        }

        // Loop until termination
        while (GameRunningFlag)
        {
            DispatchEvents();
        }

        delete pWindow;
    }
}

// Here´s where the work is done: any time the mouse moves
// within the buffered area while the window is active, invert a
// pixel and redraw the window.
// Of course, swapping in the whole buffer in response to one
// pixel change is ridiculous, but hey, this is only a test.

void CXSplatWindow::Activate(void)
{
    IsActiveFlag = 1;
}

void CXSplatWindow::Deactivate(void)
{
    IsActiveFlag = 0;
}

void CXSplatWindow::MouseDown(int x, iny)
{
    MouseDownFlag = 1;
}

void CXSplatWindow::MouseUp(int x, int y)
{
    MouseDownFlag = 0;
}

void CXSplatWindow::MouseMove(int x, int y)
{
    if (IsActiveFlag && MouseDownFlag &&
        x >= 0 && y >= 0 &&
        x < pOffscreenBuffer->GetWidth() &&
        y < pOffscreenBuffer->GetHeight())
    {
        pOffscreenBuffer->Lock();

        char unsigned *pPixel = pOff-
screenBuffer->GetBits() +
            pOffscreenBuffer->GetStride()
* y + x;
        *pPixel = ~(*pPixel);

        pOffscreenBuffer->SwapBuffer();
        pOffscreenBuffer->Unlock();
    }
}
```

Second, we'll need a `CXSplatWindow::MouseMove` that flips pixels in response to mouse movement whenever the mouse button is down and the window is active. The other `CXSplatWindow` functions set mouse down and window activation flags. To keep things easy to implement, we'll swap in the whole buffer whenever a pixel changes. In a real situation, this would be ridiculous, but for now all we have is `COffscreen::SwapBuffer`, which is good enough for this little test.

Listing 5 shows the implementation of the platform-independent application.

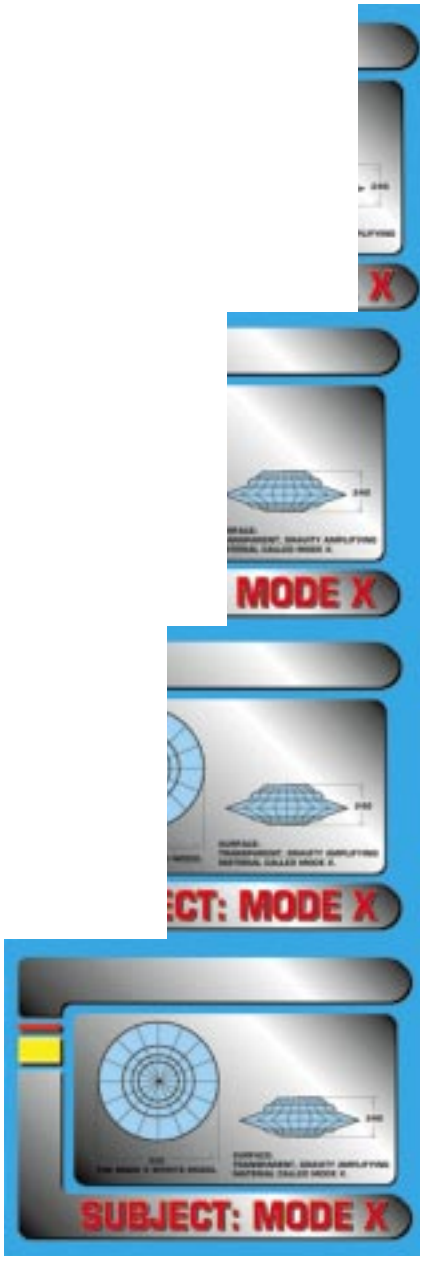
See You Real Soon

So far, we have a completely functional (though very simple) drawing application that runs on several platforms. The code to implement the specific behavior of the application used XSplat interfaces to avoid platform dependencies, and as the XSplat messaging, windowing, and double-buffering system grows beneath us, we'll be able to write more complex games that should only rarely have to know what kind of computer they're running on.

In the next article in this series, we'll address palettes and the color zero problem, take a look at some of the flaws in the design so far, and move on with the design and implementation of a real XSplat demo game. See you then! ■

After revealing his employer at the end of his last article, Jon Blossom left himself wide open to an onslaught of vicious headhunters and had to fight them off with a bamboo cane. You can reach him via e-mail at blossom@mobius.net or through Game Developer *magazine.*

The Mode X-Files



April 1, 1995, is a day I'll remember as long as I live. I was up in the gold country of California for a nice drive. It wasn't the Silicon Valley, so it suited me all the better. I pulled into an out-of-the-way filling station to top off my tank—enough to make it back to the freeway. It was getting dark, and the last thing I wanted to do was to run out of gas on a lonely back road.

The station attendant, old and simple, spoke only in binary, answering each question with "Yep" and "Nope." I handed him my 15 bucks for the lousy seven gallons of gas. That's when it all happened. An old topless Chevy Blazer skidded onto the crumbling asphalt, almost rear-ending my car. Three young men leaped out of the truck. Their faces were stone white like they had seen a ghost. One of them yanked the passenger side door open, almost pulling the muddy door off its hinges. A young woman lifelessly fell out of the seat. She was alive but limp, her eyes were open and focused on nothing.

I raced over to lend a hand. "What happened here gentlemen?" I asked as I cushioned her head with my rolled up jacket.

"We've seen it...," the driver spoke frantically.

"Seen what?" I asked. There was no answer. The young men stared at each other, looking for a reason to break some unknown secret.

Something was strange about these fellows. I walked around their vehicle, but saw nothing out of the ordinary. There were four yellow helmets in the back with California Dept. of Forestry insignias on them. The back of the truck held chain saws and shovels. A fresh dent was in the side of the truck. I looked through the driver's open door and saw a laptop plugged into the AC outlet of a cigerette lighter. I looked up at one of the young men. His fearful eyes met mine, and he looked back at the laptop. I could tell by the frightened look on his face that this was the key to whatever happened in the woods.

He watched me as I pulled the compass off the dashboard of the Blazer and held it over the computer. It spun wildly with no sense of direction to the magnetic North Pole. Amazing, I thought to myself. This can't be happening, but it is. These were young people, who but just an hour ago, believed their young selves to be immortal. But something changed us all that night. Something I never believed to be true. I was bearing witness to people who had actually had a mode X encounter of the first kind.

When I got back that night to my house. I immediately searched the internet for information regarding mode X. There was hardly any solid information—nothing to attribute to what I saw the previous night. There were documents of others who had witnessed mode X and a listing of talk shows like *Montel Williams* and *Donahue* with upcoming episodes featuring mode X encounters.

But what interested me the most were the references to a certain man—Michael Abrash—supposedly hired to reverse engineer this alien technology. Abrash had published several articles about it in credible journals. These allegedly contained detailed accounts of programming mode X. In some cases, there were references to using mode X to go beyond two

dimensions at an unearthly speed.

After more exhausting searches, again all pointing to Michael Abrash, I decided to rip apart his code and get answers for myself. My quest was clear: define the fundamentals of mode X to use in my graphics library. So I set forth on my metaphysical pilgrimage, a man, his computer, a bible on graphics, a notebook, a pencil, a scientific calculator, and some Cheetos.

The following is the result of my out-of-assembly-language experience—everything I witnessed, touched, sensed, and discovered in the couple of weeks that I locked myself in my study, unshaven but not fasting. I wish to clarify that I was a digital religious zealot and not a fanatic. When I finally cracked open my door and emerged with the answers, my nostrils flared as fresh air from the hall rushed into the cluttered room. What I discovered during this time alone is carefully documented. Some of you out there may not believe what I have seen. But I don't care. It's true. And it really happened to me.

Without further ado, I present an accurate account of information, unedited, as it happened, of the strange goings on and alien technology used in setting mode X. Oddly, this information is in no way documented by physicist Bob Lazar and in no way relates to accounts of extraterrestrial technology hidden in the Nevada desert or the alleged Sport model UFO. The logical place to start is with a prophetic question, "What is mode X, where did it come from, and what is it doing here?"

Mode X is an undocumented and unsupported video mode that uses VGA hardware efficiently in 256 colors. The Air Force, the mysterious men in black, and IBM deny its existence. Game developers stumbled on this mysterious graphics mode early on, and the most elaborate and credible witness is Michael Abrash author of these now infamous accounts.

Witnesses describe mode X as having a resolution of 320-by-240 pixels, using four 64K planes of VGA memory, and supporting page flipping. Further, a page of memory in mode X is only 76,800 pixels long. Documents released under the Freedom of Information Act describe industrial implementations using a page-flipping scheme in mode X with speeds unprecedented by the IBM-supported mode 13h. This document goes on to indicate that mode X still has adequate video memory to store a static backdrop and sprites.

Information released, based on Michael Abrash's prophetic testimony, is described herein. Places where IBM truncated Abrash's original text with heavy black markers is carefully pieced together for continuity and validity.

Untangling VGA Mode X

After placing a few phone calls, I located a credible eyewitness. I convinced the woman, who requested not to be identified, to meet me in a restaurant and discuss what she knew about configuring this mysterious mode. I convinced her that I wasn't with IBM, and as long as she divulged what she knew to me, I would get it to the public.

Fearing for her own safety, she insisted that she tell me only one of two secrets she knew firsthand—the location of the alien corpses from the 1947 Roswell incident or solid information on setting mode X. I wanted mode X. Her face went white and, with a shaking hand, she reached into her purse and pulled out a gold Cross pen.

Michael J. Norton

Fox's new advertising theme: Programming for Mode X is cool. Cool like us. Here is an account of a close encounter with this alien technology. (The truth is out there).

She glanced over her shoulder, then sketched out the following information on a napkin. She finished in under five minutes and downed her Fresca. Before her hasty flight she told me never to contact her again.

The notes from the napkin are translated here. It was a very big napkin.

Programming in mode X is tedious and not as straightforward as mode 13h. For instance, VGA can be set to mode 13h by calling the bios function. The function in Listing 1, written in Watcom C++, sets a video mode by passing it the mode value. For example:

```
setVideoMode( 0x13 );
   // sets video to mode 13h
```

Listing 1. Using Bios to Set

```
void setVideoMode( int xmode)
{
        union REGS regs;

    regs.w.ax = xmode;
    int386( 0x10, &regs, &regs);
}
```

Because mode X is unsupported by bios, you are left with the task of changing the video mode. You can achieve this by setting mode 13h with a call to bios and tweaking the video registers to the desired mode. The first objective is to remove the chain 4 attribute from video mode 13h. This is what makes mode 13h appear to be a single linear plane. The chain 4 mode can be turned off by altering bits in the VGA's sequencer registers. The reset register, index 1, and the memory mode register at index 4 need to be set as follows:

```
#define SEQC_INDEX 0x03c4
   // sequence controller index
outpw( SEQC_INDEX, 0x0604 );
   // disable chain 4 mode
```

Abrash recommends setting the clock to 25MHz for fixed-frequency monitors. This involves invoking a synchronous reset to stop the sequencer, setting the clock to the new scan rate, and restarting the sequencer:

```
#define MISC_OUTPUT 0x032c
   // miscellaneous output register
outpw( SEQC_INDEX, 0x0100 );
   // reset, stop the sequencer
outpw( MISC_OUTPUT, 0x0e3 );
   // select 25MHz dot clock and
    60Hz scanning rate.
outpw( SEQC_INDEX, 0x0300 );
   // restart the sequencer
```

The last obstacle to tackle in setting mode X is to write the proper values into the CRT controller (CRTC) registers. Writing to the CRTC registers requires having write privileges. On VGA, bits 6 and 7 of the CRTC register, vertical retrace end, are used to protect CRTC registers at indexes 0 through 7:

```
#define CRTC_INDEX 0x03d4
#define VRE_REGISTER 0x11
outp( CRTC_INDEX, VRE_REGISTER);
   // CRTC write protect bit
outp(CRTC_INDEX + 1, (inp(CRTC_INDEX +
  1) & 0x7f));
   // remove write protect
```

The write protect on various registers of the CRTC has now been lifted, so new values for mode X can be set. The values being passed are the parameters Abrash provided in his code, shown in Listing 2.

Finally, to wrap up setting mode X, the map mask register, index 02h, of the sequencer registers needs to be informed of how the memory planes are to be organized. In the case of mode X, all four planes need to be enabled:

```
outpw( SEQC_INDEX, 0x0f02 );
   // enable writes on all four planes.
```

Why didn't IBM include mode X as a supported video mode? Other video modes are plane oriented, like VGA mode 02h. In fact, mode 13h is plane oriented, but its planes are chained to make it appear as a linear address space. This feature is a good one because of the simplicity it offers in programming. However, chained planes, in this case four planes per pixel, throw off the pixel spacing. The resolution is nonsquare, where mode X provides square pixel spacing, with one plane per pixel. The aspect ratio of mode X is 1:1. Why IBM didn't offer a complimentary plane-oriented mode to 13h like mode X is certainly a mystery.

Figures 1A and 1B show chained-plane mode 13h and mode X referencing pixels. Listing 3 shows hows to set mode X with the Watcom 10.0 compiler.

Navigating Mode X

Of course, no information or key witness is valid without the testimony of an expert. I took the original notes and the napkin to a close friend at Stanford University who is

Listing 2. Abrash's Parameter Values

```
#define VALUE_VERTICAL_TOTAL       0x00d06
#define VALUE_OVERFLOW             0x03e07
#define VALUE_MAX_SCAN_LINE        0x04109
#define VALUE_VERT_RETRACE_START   0x0ea10
#define VALUE_VERT_RETRACE_LOW     0x0ac11
#define VALUE_VERT_DISPLAY_END     0x0df12
#define VALUE_UNDERLINE_LOCATION   0x00014
#define VALUE_VERT_BLANK_START     0x0e715
#define VALUE_VERT_BLANK_END       0x00616
#define VALUE_MODE_CONTROL         0x0e317

outpw( CRTC_INDEX, VALUE_VERTICAL_TOTAL );       // vertical total
outpw( CRTC_INDEX, VALUE_OVERFLOW );             // overflow
outpw( CRTC_INDEX, VALUE_MAX_SCAN_LINE );        // set for double scan
outpw( CRTC_INDEX, VALUE_VERT_RETRACE_START );   // vsync start
outpw( CRTC_INDEX, VALUE_VERT_RETRACE_LOW );     // v sync end and protect
outpw( CRTC_INDEX, VALUE_VERT_DISPLAY_END );     // vert displayed
outpw( CRTC_INDEX, VALUE_UNDERLINE_LOCATION );   // turn off dword mode
outpw( CRTC_INDEX, VALUE_VERT_BLANK_START );     // v vblank start
outpw( CRTC_INDEX, VALUE_VERT_BLANK_END );       // v blank end
outpw( CRTC_INDEX, VALUE_MODE_CONTROL );         // turn on byte mode
```

Figure 1a. VGA Mode 13h 64K Linear Bitmap Organization

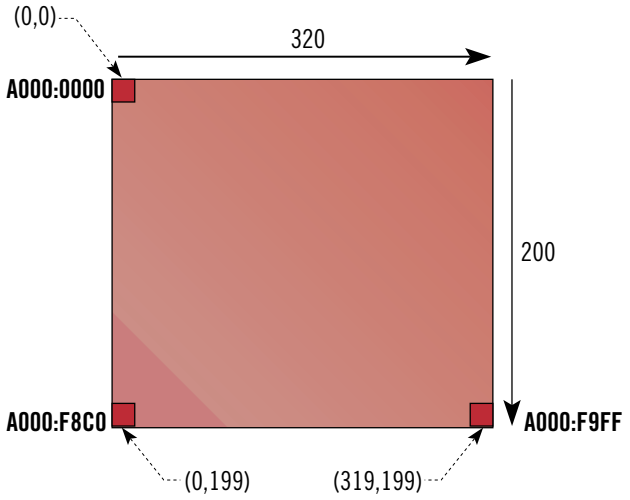
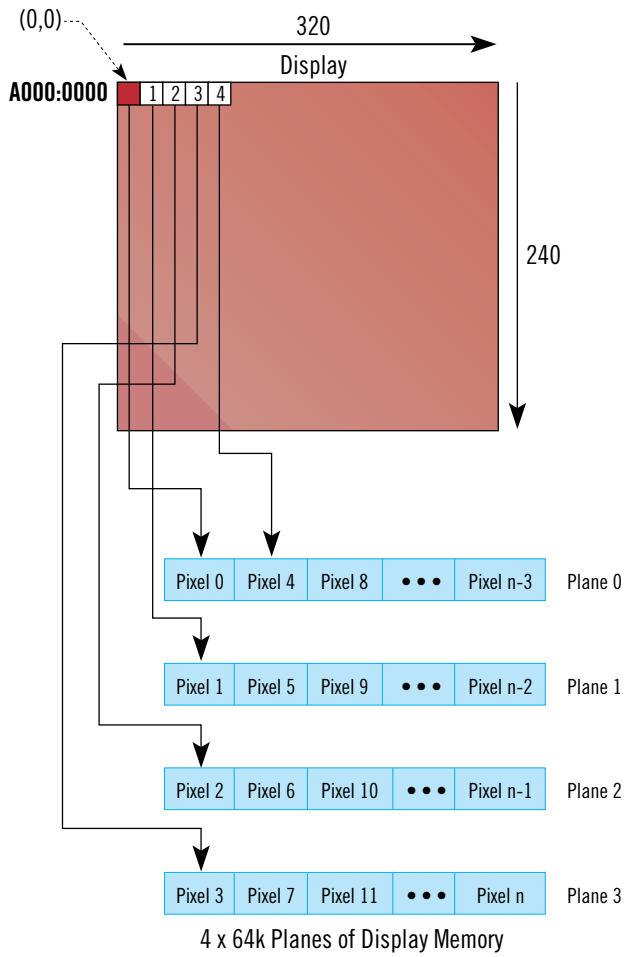


Figure 1b. VGA Mode X Plane Memory Organization



part of the SETI project. He took one look at the notes and sat down in awe. He told me that while I was waiting I could take a peak at some data of what he believed to be signals from an extra-terrestrial civilization.

I told him I was not in the least bit interested and to concentrate on the information I had provided him. By 2:00 the following morning—and three large combo pizzas later—we staggered over to an XWindows terminal logged into a Cray supercomputer. Characters of alien origin shot across the screen. What does this all mean I asked myself. My affiliate reached into his drawer and pulled out a wire hanger wrapped in aluminum foil. He informed me to stick my head out the window and hold the hanger still on his mark.

"Now," he yelled as he scrambled to dump the file to a print queue.

By now, its obvious that working in the mysterious mode X is not as simple as its supported, chained planar video mode 13h is. Plotting pixels in a linear bitmap is easy. But now that the attributes of the video memory have been modified and its chained mode disabled, navigating this alien mode can be difficult.

Mode X and mode 13h do share two commonalities: that they're both one byte per pixel and they both have 320 columns per row. The latter is true because mode X is a hybrid video mode of 13h and only the scan lines were reconfigured.

To write a pixel, a pointer must be initialized to reference video memory. This example uses DOS4/GW and C++, so the video pointer will not be type far. In protected mode under DOS4/GW, the video memory is mapped in the same segment as the code. To set a pointer to VGA memory, be it mode 13h or mode X, we use the following snippet of code:

```
char *pVideoMem;
   // pointer to VGA memory, DOS4/GW
pVideoMem = (char*)(0xA0000);
   // initialize pointer to VGA memory.
```

DOS4/GW uses physical memory addresses; that's why the pointer was initialized to location `0xA0000` and not segment `A000:000`. This pointer references the first byte in VGA memory at offset 0. In plain English, this is the first pixel in the upper left-hand corner of the screen. The pixel can be plotted by assigning the location a byte value:

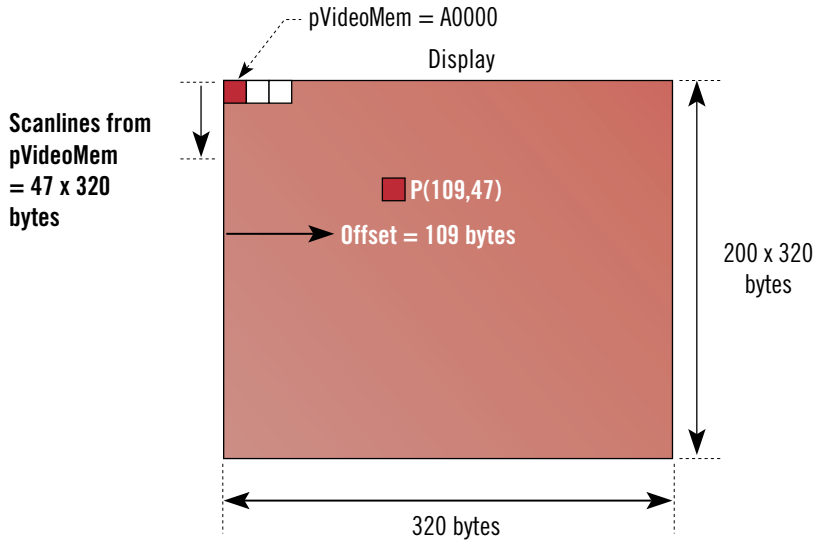
```
char pixelValue;
   // a value to write to memory
pixelValue = 127;
   // assign a value
*pVideoMem = pixelValue;
   // write the value to memory
```

This code is useful if we only write to the first coordinate of the screen, but what about the tens of thousands of other pixels we wish to set? To plot other pixels, we will need to understand how video memory is organized.

In mode 13h, video memory is chained to give it a flat bitmap effect. The plotting of a pixel in one of the 320 columns by 200 rows is a simple matter of arithmetic. For instance, to plot a point, P(109,47), where X=109 and Y=47, we simply calculate the scan line and the byte offset into the scan line of the location we wish to plot.

To compute the scanline (or row) from a given point, P, multiply the Y coordinate times the maximum number of bytes per scanlines. This value will tell us at what logical row in memory the pixel is

Figure 2. Pointer Arithmetic, Point Coordinates, and Mode 13h



located. From this location, the pixel will be at an offset X from the start of the row. Sounds a little confusing? Let's figure an example.

Find the pixel location in VGA memory of P(109,47) in mode 13h:

```
scanline = Y *320 = 47 *320 = 15040 =
  0x3AC0
offset = X = 109 = 0x6D

pixel memory
location      = VGA memory ptr + scanline
                + offset
              = A0000 + 0x6D + 0x3AC0
              = 0xA3B2D
```

The code looks like this:

```
unsigned int scanLine, offset;
scanLine = Y * 320;
   // compute the scan line, Y
*(BYTES/ROW)
offset = X;
   // offset of pixel from scanline
pVideoMem = (char *) (0xA0000) + scanLine
+ offset;
   // ptr to pixel location
*pVideoMem = pixelValue;
   // assign a value to the pixel
```

Figure 2 shows the pointer arithmetic and point coordinates for this example.

Navigating mode 13h is easy. Now with the understanding of how a chained linear bitmap works, we can apply what we know to unchained planar memory, such as mode X. The important thing to keep in mind is that mode X scanlines are not 320 bytes per row. Mode X is unchained—the bytes are organized across four planes of memory. So a mode X scan line now has (320 bytes/scan line) / 4 planes = 80 bytes / scan line plane.

The computation of a scan line in mode X, using the example P(109, 47) looks like:

```
scanLine = Y * 80;
   // mode X scan line
```

Figure 3 shows the pointer, planes, and points in mode X.

This value is useless unless we can tell the VGA which memory plane this scan line corresponds to. The plane can be determined by playing a few games with the X coordinate using logical operators:

```
X = 109 = 0x006D = 0000 0000 0110 1101
```

We want to filter and mask out the X value:

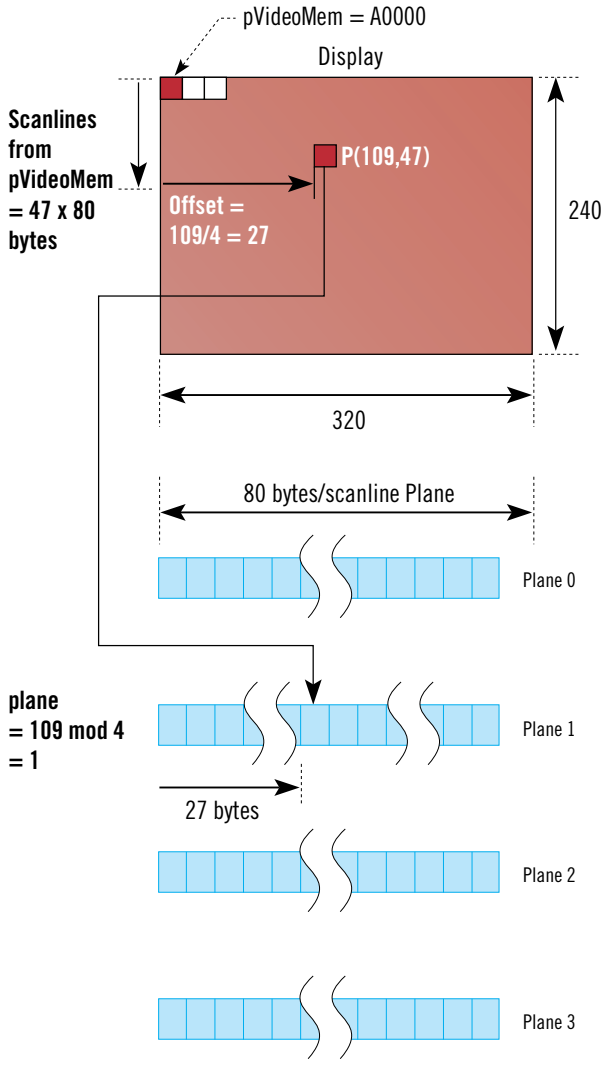
```
X = 109 0000 0000 0110 1101
& MASK = 255 & 0000 0000 1111 1111
Result= 109 0000 0000 0110 1101
```

The operation is very uneventful and is in place only to act as a bit filter. We want to be certain that we accurately grab the low order eight bits. Actually, the low-order two bits are the only ones of concern because these will reveal which memory plane this scanline belongs to. Masking out the plane is also accomplished with a logical AND operation.

Listing 3. Setting Mode X with the Watcom 10.0 Compiler

```
void setModeX(void)
{
        setVideoMode(0x13);                     // set the bios supported video mode 13h

        outpw( SEQC_INDEX, 0x0604 );            // disable chain 4 mode
        outpw( SEQC_INDEX, 0x0100 );            // reset, stop the sequencer

        outp( MISC_OUTPUT, 0xe3 );              // select 25 Mhz dot clock and 60 Hz scan rate
        outpw( SEQC_INDEX, 0x0300 );            // restart the sequencer

        outp(CRTC_INDEX, 0x11 );                                    // vertical retrace end register
        outp(CRTC_INDEX + 1, (inp(CRTC_INDEX + 1) & 0x7f));         // remove write protects

        outpw( CRTC_INDEX, VALUE_VERTICAL_TOTAL );                  // vertical total
        outpw( CRTC_INDEX, VALUE_OVERFLOW );                        // overflow, bit 8, vert counts
        outpw( CRTC_INDEX, VALUE_MAX_SCAN_LINE );                   // set for double scan
        outpw( CRTC_INDEX, VALUE_VERT_RETRACE_START );              // vsync start
        outpw( CRTC_INDEX, VALUE_VERT_RETRACE_LOW );                // v sync end
        outpw( CRTC_INDEX, VALUE_VERT_DISPLAY_END );                // vert displayed
        outpw( CRTC_INDEX, VALUE_UNDERLINE_LOCATION );              // dword mode off
        outpw( CRTC_INDEX, VALUE_VERT_BLANK_START );                // v vblank start
        outpw( CRTC_INDEX, VALUE_VERT_BLANK_END );                  // v blank end
        outpw( CRTC_INDEX, VALUE_MODE_CONTROL );                    // turn on byte mode

        outpw( SEQC_INDEX, 0x0f02 );   // enable writes on all four planes

}
```

Figure 3. Pointers, Planes, Points, and Mode X



There are four planes ordered 0 to 3:

```
PLANE MASK = 3 = 0x03 = 0000 0000 0000
0011

Result = 109 0000 0000 0110 1101
& PLANE MASK= 3 & 0000 0000 0000 0011
Plane = 1 0000 0000 0000 0001
```

This operation tells us which plane of display memory we are going to write a value to. The sequence controller `MapMask` register needs to be instructed which plane this is to allow the CPU to carry out a write operation on this memory. The `MapMask` register is an 8-bit register, and only the four low-order bits, 0 to 3, are of any significance. Bits 0 to 3 correspond to planes 0 to 3. Setting bit 0 allows write operations by the CPU to memory plane 0. This behavior is identical for all four bits in the `MapMask` register at index 02h.

The code for this operation looks like this:

```
nPlane = (x & 0x00ff)
  & 0x03;
/* calculate the
plane from the X
coordinate*/
regs.w.ax = 0x0100 +
  0x02;
/* set the index to
02h and initialize
bit for pixel´s
plane*/
```

Pay careful attention to the last line of code. This is not computing an index into the sequence controller. It's actually a clever scheme to initialize the `AX` register before calling the `outpw()` function. The low byte, `AL`, is being initialized to the index 02h, the `MapMask` register in the sequence controller. The high byte, `AH`, is being initialized to set bit 1 of the `MapMask` register:

```
regs.w.ax = 0x0100 + 0x02 = 0x0102 =
  0000 0001 0000 0010
index of Map Mask register = AL = 0000
  0010
initialize pixel plane to 1 = AH = 0000
  0001
```

The value in `AH` is then shifted left `nPlane` times. This value will be loaded into the `MapMask` register to enable a write to the desired plane of video memory:

```
regs.h.ah = regs.h.ah << nPlane;
   // set the bit for the pixel´s plane
outpw( SEQC_INDEX, regs.w.ax );
   // set Map Mask to enable write
```

The write to the memory plane can now be performed in the familiar manner. The one exception to this operation is that X is no longer a linear mapping. The offset is now at X/4. So:

```
scanLineOffset = (y*80) + (x>>2);
pVideoMem = (char*)(0xA0000 +
  scanLineOffset);
   *pVideoMem = pixelValue;
```

That's it for writing a pixel to a mode X planar memory location. You can now port your old mode 13h line drawing and graphics libraries over to mode X. The complete listing is shown in Listing 4.

By now, both my colleague and I had had very little sleep but we were eager, nonetheless, to test this unearthly technology. We loaded up a laptop and drove out to a dry lake bed in the Nevada desert where we executed the following test program. The test was short. We merely powered up mode X, threw out a rectangle on the laptop's display, and powered it down. I can attest to the fact that my adrenaline was racing, for we had no idea what mode X would do, nor to the destructive power it was capable of. The test was successful, and as UFOlogist Bob Lazar would say, uneventful. The source code used at the Nevada test site is shown in Listing 5.

Copying Linear Bitmaps to VGA Planar Memory

My scientific colleague made some good assumptions about how to use mode X, but it was my radical cyberpunk anarchist friend who made the most astounding contributions. Based on the frequency of cattle mutilations per year divided by the rate that the giant stone heads on Easter Island creep toward the sea, the following piece to the big puzzle came to light. And I quote, "This technology could be radically applied to games, dude!" What revelation had my reality-twisted friend stumbled upon? What could this possibly mean? And why did he have a "Linux Inside" sticker stuck to his tower case?

Plotting pixels is okay for tinkering

with mode X, but let's face it, we want to get down to some serious business. Games must have cool backgrounds and sprites, right? The only way this can be accomplished is if .PCX and .IFF files can be loaded into display memory. These files are typically decoded into memory and are suitable for a display mode like 13h. Now you're going to learn how you can expand your horizons and your graphics library at the same time. The goal here is to write a useful tool that takes the decoded graphics image from linear memory and writes it to display memory for use with mode X.

A good starting point is to identify the differences in linear memory and VGA planar memory. A Deluxe Paint LBM image in memory is a 320-by-200 pixel bitmap with 256 colors. Describing this image in rectangular terms we have:

- Left = 0
- Right = 320
- Top = 0
- Bottom = 200.

Images are commonly described in terms of logical rectangular coordinates. In actuality, they are stored as chunks of memory, where their scan lines are stored consecutively, one right after the other. For the programmer and game designer, it is simpler to maintain descriptions of images in terms of the rectangular boundaries. It is easy to determine physical memory attributes of a source image from a descriptive bounding rectangle. Using the example of a Deluxe Paint II image, the physical attributes of memory can easily be retrieved. This pseudocode calculates the number of bytes per scanline and starting offset of the image in the chunky bitmap:

```
unsigned int srcBitMapWidth, left, top,
  right, bottom;
srcBitMapWidth = right - left = 320 - 0
  = 320
srcOffset = (srcBitMapWidth  * top)
  + left = (320 * 0) + 0 = 0
```

These are precisely the values we would need if we were to write this bitmap to VGA mode 13h memory. In fact, all that's needed from here is a pointer to VGA memory and a pointer to the source image. The entire image could simply be blasted to the screen with a call to `strcpy()` because the source is 320 by 200 bytes and so is the destination VGA memory.

That's basically all we would have to do if we were working in video mode 13h. However, we're working with mode X. We must now calculate a destination bitmap based on four memory planes. The difference between the source image memory and destination video memory is that the image is stored in linear memory, and the destination is planar video memory. Our task is to devise a means to map a chunky bitmap in memory to four VGA memory planes.

The description of the source image in terms of `srcBitMapWidth` and `srcOffset` needs no further attention. This is all the information, including the image `srcPtr` itself, we will need to map the chunky bitmap. Similar descriptions are needed for the destination memory. The destination bitmap width for VGA plane memory is simply the source bitmap width divided among four planes of VGA memory. With the value of `srcBitMapWidth`, `destBitMapWidth` and `destBitMapOffset` can be computed as follows:

```
destBitMapWidth = srcBitMapWidth / 4
destBitMapOffset = (destBitMapWidth *
  top) + (left/4)
```

If you're still in the game here, you've determined by now that this information sheds no light as to which plane of memo-

Listing 4. Plotting a Pixel in Mode X

```
// plot pixel in mode X
#define BYTES_PER_SCAN_LINE 80
#define MAP_MASK 0x02
#define INITIAL_PLANE_BIT 0x100
#define MAP_MASK_PLANE_INIT INITIAL_PLANE_BIT + MAP_MASK
#define BIT_FILTER_MASK 0x00FF
#define MAX_PLANE_N 0x03

void writePixelModeX( unsigned int x, unsigned int y, char pixelValue)
{
        union REGS regs;
        char *pVideoMem;
        char nPlane;
        unsigned int scanLineOffset;

        nPlane = 0;
        scanLineOffset = 0;

        scanLineOffset = (y*BYTES_PER_SCAN_LINE) + (x>>2);
        nPlane = (x & BIT_FILTER_MASK) & MAX_PLANE_N;
        regs.w.ax = MAP_MASK_PLANE_INIT;   // set index 02h, and initialize plane = 1
        regs.h.ah = regs.h.ah << nPlane;
        outpw( SEQC_INDEX, regs.w.ax );
        pVideoMem = (char*)(0xA0000 + scanLineOffset);
        *pVideoMem = pixelValue;
}
```

Listing 5. Test Site Source Code

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

#include "modex.hpp"

void main()
{
  unsigned int i;

  setModeX();

  for (i=0; i<240;i++)
  {
     writePixelModeX(0,i,0x23);
     writePixelModeX(319,i, 0x23);
  }

  for (i=0;i<320;i++)
  {
       writePixelModeX(i,0,0x23);
       writePixelModeX(i,199,0x23);
       writePixelModeX(i,239,0x23);
  }

  getch();
  setVideoMode(0x03);
  printf("done... press any key to
  continue\n");
}
```

ry we are to enable CPU writes to. Recall from the earlier discussion of plotting pixels in mode X that the X coordinate was the key in unraveling the mystery of which plane is to be write enabled. Well, the bitmap doesn't have an X coordinate or...does it? Simple high school geometry proofs showed that a rectangle can be described by two points: P1(x1, y1) and P2(x2, y2). The bounding rectangle of the bitmap is defined by two points. They are P1(left, top) and P2(right, bottom). The left boundary is what we'll use to determine the initial plane to write enable:

```
char nPlane, planeMask;
nPlane    = left & 0x03;
left      = 0000 0000 0000 0000
& mask    = 0000 0000 0000 0011
nPlane    = 0000 0000 0000 0000 = 0

planeMask = 0x11 SHL nPlane = 0001 0001
<< 0 = 0001 0001
```

By performing a logical AND operation on the left coordinate with the mask 0x11, we can determine the initial plane we need to enable. Play around on paper and shift the bits 0x11 about two or three times. Computing the initial plane is significant for two reasons. First, you do not need to have a rectangle that's 320 by 200 or starts at left = 0, top = 0. Second, this initial value will come into play later, for cycling the planes to write enable.

But that's coming up, and let's not get ahead of ourselves. With this information in hand, we now know the plane for the first destination pixel. The following code does this:

```
nPlane = left & MAX_PLANE_N;
planeMask = 0x11 << nPlane;
```

The first plane is now calculated, but not yet enabled. We must loop through the entire chunky bitmap image in memory and write it to the proper VGA memory planes. This will require a nested loop—the outer loop will walk the rows of the linear bitmap, and the inner loop will handle translating the linear image scan lines to the VGA memory planes. The pseudocode is shown in Listing 6.

This code resembles a brute-force mode 13h blitter. The exception is the initialization of register bitsROLplane with the current planeMask value we computed earlier. The code then drops inside to the nested loop where a miracle occurs RectWidth - 1 times.

Now what's so miraculous about the inner loop? It needs to do several things:

- Enable the desired plane for a write operation
- Change the plane for the next pixel
- Copy the source image pixel to the proper VGA plane memory location.

The first step is simple and is already included in the pseudocode in Listing 6. The plane, as you already know, is enabled by setting the proper bits in the MapMask register of the sequence controller. That's why the outp() function call is in the inner loop. The second step, involving changing the plane for the next pixel is not so easy.

Listing 6. Pseudocode for a Nested Loop

```
RectWidth = right - left;
nRows = bottom - top;

outp( SEQC_INDEX, MAP_MASK );
for(rows=0; rows<nRows; rows++)
{
        bitsROLplane = planeMask;
        for(scanlines=0; scanlines<RectWidth; scanlines++)
        {
                outp( SEQC_INDEX + 1, bitsROLplane );  // enable desired plane for a write
                A miracle occurs here...               // occurred sometime after the 7th day
        }
        pVideoMem += destBitMapWidth;
        SourcePtr += srcBitMapWidth;
}
```

Instead of computing the plane by looking at the current offset, we will approach the problem from a simpler angle. The plane to be enabled is cyclic by design. That is, if plane 2 is being enabled for a write operation, the next plane to be enabled after the write is plane 3. After plane 3 has been written, we need to wrap around back to plane 0. We already know the initial plane to enable. We calculated this earlier. This value will initiate the cyclic sequence. By taking this initial value, planeMask, we can determine the next plane to be enabled by performing a ROL operation. You can access an example of this cycle on the *Game Developer* ftp site (ftp://ftp.mfi.com/gamedev/pub/src).

In the final ROL operation on the planeMask, the bit in nibble 7 moved left out of the byte and set the CPU CarryFlag. In a ROL operation, when the CarryFlag is set, the bit that rolled out of nibble 7 rolls back into nibble 0. This emulated ROL operation on the planeMask bits provides a simple mechanism for cycling the planes to write enable. The only drawback is that ROL operations are not provided in the C library function calls or by any binary operators. So we need to write one ourselves:

```
unsigned int bitCarryFlag = 0;
char bitsROLplane = planeMask;
```

The operation of performing a ROL on a byte value can be accomplished by using the C and C++ shift left operator, <<. This will perform a bitwise shift left, but will not indicate when a bit set in nibble 7 has shifted out to the left:

```
bitsROLplane = bitsROLplane << 1;
   // shift bits left once
```

The bitwise shift left is basically the SHL operator. Once the bit has shifted out of the high nibble, it falls into the bit bucket never to be recovered. We must manually check the bit in nibble 7 before the logical shift operator is used. The nibble can be examined with a mask operation, which we'll use to mimic a CarryFlag when the bit shifts out of the byte. The following example illustrates a case where plane 3 wraps around to plane 0, based on the result in the bitCarryFlag:

```
bitCarryFlag = bitsROLplane & nibbleMask;

bitsROLplane = 10001000
nibbleMask= & 1000 0000 = 0x80
bitCarryFlag = 1000 0000 = 0x80
```

Now perform the logical SHL operation on bitsROLplane:

```
bitsROLplane << 1 = 0001 0000
```

This is the condition we need to test for to wrap the plane back to zero. When the value in bitCarryFlag is equal to 80h, the plane will wrap. This test carried out in the following manner:

```
if (bitCarryFlag == 0x80)
   wrapPlane;
```

What precisely does it mean to wrap the plane? The value 0x80 is a bit filter to alter the contents in bitsROLplane to fit the cyclic model. The wrapping around occurs by performing a logical OR operation on bitsROLplane:

```
bitsROLplane = bitsROLplane | 0x01;
bitsROLplane = 0001 0000
OR             0000 0001
bitsROLplane = 0001 0001
```

That's the whole tamale for plane wrapping. It's not so difficult once you've walked through it. The code for performing this operation looks like this:

```
bitCarryFlag = 0;
bitCarryFlag = bitsROLplane & 0x80;
   // test for non-zero high order nibble
bitsROLplane = bitsROLplane << 1;
   // set mask for next pixel´s plane
if (bitCarryFlag == 0x80)
   // if high order nibble non-zero
  bitsROLplane = bitsROLplane | 0x01;
   // carry the bit to lowest nibble
```

The third and final dilemma we need to resolve is mapping linear bitmap bytes to the corresponding VGA planes. The outer loop controls what row is being updated on both the bitmap and the VGA memory. The inner loop is walking the scanlines. This is simply moving to the offset from the start of the scanline and updating each pixel in the bitmap to the designated plane of memory from the ROL operation.

Translating offsets in mode X memory is simple. Remember that we treated the X coordinate in the earlier examples as the offset in memory. The coordinate X, or the offset, mapped to a VGA plane at offset X/4. The mapping of a byte from a chunky bitmap to a defined VGA plane of memory is the same. Translating pixels is accomplished by the following code snippet:

```
pVideoMem[scanlines>>2] =
  SourcePtr[scanlines];
```

The pointer pVideoMem is referencing the VGA memory plane, and SourcePtr is referencing the byte to be mapped. The index, scanlines, is byte offset in memory from SourcePtr of the pixel to be mapped:

```
pVideoMem[scanlines>>2] =
  SourcePtr[scanlines];
```

Trust No One

Now you have enough information to be dangerous. No longer can the assembly language listings in Michael Abrash's *The Zen of Graphics Programming* (Coriolis, 1994) be of any threat to you. If you don't have this book yet, buy it! Abrash gave away all the industry secrets in one book. As I mentioned, the source code is old, real mode 8086 code, and requires some attention when porting it to Watcom protected-mode compilers.

As I paced the small apartment of my cyberpunk anarchist friend, my deep probing meditations were disrupted by a stereo tone from the Sound Blaster 16 card of his computer, signalling the succesful end of a compile.

He had compiled the linear bitmap translator and was ready to execute it. Time flowed in slow motion as I ran towards him, shouting, "Nooo!" My hands reached out to stop his right index finger as it glided over to tap the enter key on his keyboard.

His head turned slowly, grinning like some sinister goblin, his eyes hidden behind round purple reflective lensed cyberpunk sunglasses. The words rolled off his tongue with no concern to their impending dangerous implications.

"Rock and roll, dude!" he said, and his finger pressed the enter key. I froze in my tracks as the computer set the alien mode and wrote the bitmap in system memory out to the mode X display. My heart pounded harder and shorter than it did out in the Nevada desert. I looked at him, my facial expression scorning him for being so technically irresponsible. The consequences could have been far worse. The thought that raced through my mind was that mode X could somehow chaotically react with our atmosphere and burn up all the oxygen. But that didn't happen.

"Whoa, dude," he said. "Check it out—we're all still here." His hands pressed firmly against the chest of his tie-dyed t-shirt. He wanted to make certain that the molecular organization of his body was still structurally intact. He got up from his chair and pulled back the curtain. "Excellent!" he nodded, "The time continuum seems to be in order as well."

This test somehow alerted the boys from Big Blue. A network tap started to set off an alarm. They were getting a fix on our location. The anarchist cyber dude started scrambling his IP octets to confuse the people tracking his network routes and hops.

"Until we can finish constructing our They Live VR visors, no one is safe," he gravely stated.

Due to space constraints,I couldn't list all the code necessary to this encounter. You can access the complete listing for copying a chunky bitmap to VGA mode X planar memory and the code used in the test by the cyber dude on the *Game Developer* ftp site.

All I can say for now is, watch out. Mode X works—on Windows 95, too, in protected mode. Maybe Redmond is where we'll find them. You don't have to believe me, see for yourself. Mode X is out there and it's in your system. ■

Michael J. Norton holds a B.S. in physics and has worked as a programmer for 12 years. He is currently working on a book for programming the Windows 95 SDK.

Gamin' for Grrrrls

Barbara Hanscome

How have some game companies tried to appeal to girls? Are these games vastly different from those designed for a more general audience? And how?

How do we make games girls and women will buy? It's a question today's game companies must ask themselves or lose out on a little more than half the potential electronic entertainment market. Most game industry executives seem to conclude that if they are to consider targeting girls and women at all, it will not be at the risk of excluding their core market of boys and men, which is why games for kids are often touted as being "gender inclusive." But what does gender inclusive mean, exactly, and is it the most effective way to attract the female market? Game industry pros have conflicting opinions.

Some feel that the games claiming to be gender inclusive really aren't, and that all this talk of gender-inclusivity just perpetuates serving the needs of a male audience. "Until the day when 'gender neutral' stops meaning 'really boys,' I truly believe that we have to create games specifically for girls," says cartoonist Trina Robbins, who wrote and designed Sanctuary Woods' interactive CD-ROM Hawaii High, Mystery of the Tiki.

If you look at some of the research into gender differences, the idea of a gender inclusive game seems like a difficult—if not impossible—goal to reach. A 1984 edition of *Personal Computing Teacher* features a study "Situational Stress as a Consequence of Sex-Stereotyped Software," which states that if you take games with the characteristics girls like and make boys play it, boys show signs of situational anxiety and their performance and focus deteriorates. The study claims girls have the same reaction when playing with "boy" software. (At the same time, a common belief exists in this industry that girls will play "boys' games," but boys will not play "girls' games"—an attitude that can't do much except tilt the scales toward the tastes and preferences of males.)

Still, game developers are cautious about stereotyping and generalizing gender differences in game preference and play. "If we say 'girls don't like violence so we'll give them Barbie Dolls' or ' girls want story, so we won't let them play Sonic the Hedgehog,' we're creating paradigms of exclusion for both boys and girls," says Margy Gilman, an interactive TV producer in San Diego. "It's just as discriminating to boys [to give them only violence] and not to give them story and character. There are lots of boys who care about that."

Creative Wonders is also cautious about excluding boys. Its new title Madeline and the Magnificent Puppet Show boasts a famous female heroine chosen for her pluckiness and popularity with girls and women from the storybooks of Ludwig Bemelmans. But Creative Wonders says Madeline is not a game "for girls," but a game for children that girls "especially" will like. Even the Women's Interactive Entertainment Association has taken a stance supporting "gender inclusivity" over "gender specificity." "It's not about making gender specific titles," explained WIEA president Valerie Hennigan. "It's about making them 'gender friendly'....and it's marketing those games to girls. Interactive entertainment isn't even marketed to females."

While the discussion continues about how to best reach girls and

women—be it with an affirmative action approach or with gender inclusive games for boys and girls—this installment of Chopping Block takes a look at a few titles for children and what characteristics might make them appeal to girls (and in some cases, to boys as well).

Sega and Nintendo Games

Much of the research into girls' game preferences suggests that what many of the classic arcade, Nintendo, and Sega style games are made of (repetition, scoring, speed, violence, target shooting, time limits, reinforcement based on skill instead of effort) is unappealing, boring, and senseless to girls.

In 1994, Sega introduced Crystal's Pony Tale, a game designed to attract girls aged four to seven to the Sega platform. The game was created by Sega's all-women Girls Task Force to be a girl's first Sega game—and in many ways that is exactly what it is (kind of a Sonic the Hedgehog with pink training wheels.)

In Crystal's Pony Tale, Crystal's friends have been kidnapped by a wicked witch and hidden throughout the game. Players must help Crystal find seven crystal shapes to free her pony pals from captivity. Using the Sega keypad, players help Crystal gallop and leap in the air and capture keys and horseshoes. The horseshoes let Crystal move to three other "levels" or environments in the game, and the keys let her open treasure chests, where crystals and clues can be found. The hidden ponies are marked in the game with a silhouette of a horse and the shape of a crystal. If Crystal has the correct rock in tow, the player can move Crystal next to the pony's shape, press the action key, and release Crystal's friend.

Crystal must collect the right number of horseshoes before she can pass into the next environment, but the game isn't based on scoring or attaining the next level as much as it is on helping Crystal help her friends—a goal a four-year-old girl might find more worthwhile than shooting at a target. When Crystal succeeds in doing this, hearts of love and gratitude spring forth from her pony friend onto Crystal's head. It's corny, but the positive feedback can't be ignored.

Still, you certainly wouldn't call Crystal's Pony Tale a "story-driven" game—and research indicates that girls are more interested in story lines than fast action. Crystal's Pony Tale follows the same tried-and-true, linear action formula of any Sega title. The "story" is more akin to a game "theme."

The game is slower paced than your typical Sega or Nintendo action adventure. There is no time limit; Crystal can stop and talk to a cow in the barn, munch on hay, or sniff sunflowers without the threat of being bonked or killed by incoming villains. The player also has a game map of sorts, which enables Crystal to move between levels if she wants to.

In keeping with research showing girls are more motivated by sound, voices, and music than boys, the game lets girls choose the background music at the beginning of the game from a number of classical pieces.

As for violence, well, Crystal isn't a exactly Steven Segal, but she does exhibit some aggressive behavior. Whenever Crystal tries to rescue a friend, the pastel-colored sky turns dark, the music picks up tempo, and the evil witch swoops onto the scene. If you continuously press the action key, Crystal rears on her hind legs and whinnies repeatedly. Without bloodshed or bullets, the witch disappears and serenity returns.

Character involvement is crucial to a girl's enjoyment of a game experience, according to research. But we know very little about this pony. The players' guide tells us that Crystal is the "shyest" pony in the land, but if you don't read the guide, you miss out on this information (which might give some players a sense of accomplishment after helping Crystal overcome her shyness and fight the witch).

Sega seems to prefer using animals in their gender-inclusive children's games instead of humans, perhaps because they are easier to gender-neutralize. In fact, Crystal could be either male or female; the character is so poorly rendered you can barely see its face. If I were a five-year-old girl and my big brother were playing a richly animated game like Earthworm Jim on our family Sega sys-

WHAT DO GIRLS LIKE? WHAT DO BOYS LIKE?

Industry and academic research reveal some interesting differences in what many girls find fun, challenging, and engaging in electornic entertainment. But of course, as you'd expect when dealing with human preferences and tastes, contradictions abound. Still, some of this research is hard to ignore. Here's a summary of some of the findings:

- **Females prefer utility, reality, and constructive goals, while males prefer aggressive and competitive themes and are less interested in the far-reaching benefits of a game.**
- **Females find sound, especially voices, motivating, while males are often observed playing with the sound turned off.**
- **Females are less interested in "games" (meaning play involving established rules, clear objectives, and the notion of winning) than other forms of play.**
- **Females like interactive products without scoring or competition and are less interested in winning. Males often mention scoring (and winning) when discussing games.**
- **Females are flexible with rules and will change rules to suit a situation, while males are more inflexible with rules. They use rules to determine domination and enjoy following them.**
- **Females are frustrated when having to start over from the beginning of a game. Males find this motivating.**
- **Females are more confident, optimistic, and motivated when feedback focuses on effort vs. ability. They see feedback as a reflection of their ability and will question themselves if the feedback is negative.**
- **When feedback is based on effort, males see it as a reflection of poor performance and loose motivation.**

tem, I'd feel a little slighted.

Some girls, especially girls at the older end of the game's age-range, might find the Arthurian castles a bit tired and cliché. According to Sega, they set the story in this environment because research indicated girls like fantasy, but does fantasy really mean witches and goblins? It's no wonder many of them skedaddle on over to the more exciting world of Donkey Kong Country—a game popular with both girls and boys.

Donkey Kong Country

Nintendo's Donkey Kong Country wasn't designed "for girls" or with girls in mind, but girls certainly like it. In a nationwide survey of 1,600 kids taken by interactive TV producer Margy Gilman, Donkey Kong was listed in the top five favorite games of girls (along with Mario Brothers, Sonic the Hedgehog, Lion King, and—believe it or not—Mortal Kombat).

The game takes players on a jungle adventure with two primates: Donkey Kong (a big macho ape) and his friend Diddy (a small, agile monkey). Your goal is to help them recapture their horde of bananas, which have been stolen from them by the evil Kremlings. You help Donkey and Diddy run and jump over obstacles, grab ropes and swing through the forest, even swim through water worlds capturing bananas, letters (that spell out Donkey Kong of course), and extra lives. You've got to move fast or any number of beasts will run you down. When you "lose a life," you start over from the beginning of the level; if you lose several lives in succession, you "die" and start over from the beginning.

Going by the research into what girls like in games, there would be little in Donky Kong to get a girl hooked on video gaming. The game isn't based in reality; nor does it involve a story. (You're supposed to be helping Donkey and Diddy regain their hoard of bananas, but if you don't read the players guide, you don't know this or care.) Game feedback is based on the number of points you score and the level you achieve—which research says girls find less motivating than feedback based on trying. So why is the game popular with girls? Could it be that a large number of girls out there enjoy the fast action, the funny animations, and three-dimensionally rendered jungle worlds, and the challenge of attaining points and mastering the levels?

Unfortunately, Candy Kong, the only female character in Donkey Kong Country, serves as little more than a showpiece.

What I find disappointing in Donkey Kong is that the game features only one female character: Candy Kong, an ape in a hot-pink bathing suit with nothing better to do than pose provocatively in a booth and help Donkey and Diddy save their points. Basically, she's stuck playing Vanna White while the guys go out for a swingin' banana hunt. Humph! That's not very "gender inclusive."

Another gripe: you're deprived of most of the game until you reach a certain level of skill. Donkey Kong contains seven environments and 40 levels. Each level is wonderfully different and features new obstacles and characters, which you'll never see unless you play the game well enough and score high enough to reach these levels. Game developers I've spoken with call this setup "having to earn the game," and they say girls find this game plan more of a turn-off than blood spurting from a corpse any day.

Sure, girls can master the game just as well as boys can, but research suggests that they'll become bored (or frustrated) with the experience long before that will happen.

Donkey Kong Country 2 coming out this December, might have more "girl appeal." Girls might find the goal of the game more "worthwhile": you're actually trying to save Donkey Kong from capture by the evil Kremlings. But more important, it stars a female character (at last, a female saving a male in distress!). This game stars Dixie Kong, Diddy's little monkey friend, who sports a pink t-shirt and a long, platinum-blonde ponytail. Some might argue that she's really just Diddy in drag, but it's hard not to smile when watching her in action. Face it, she's cute. And she takes the lead throughout the game, dragging Diddy around and leaping above obstacles, her ponytail whipping about like a propeller. According to a Nintendo spokesperson, the company hopes Dixie will be the next flagship character for the Nintendo line.

Hawaii High, Mystery of the Tiki

Released in 1993, Hawaii High, Mystery of the Tiki, from Sanctuary Woods was one of the first interactive CD-ROMs created specifically for girls aged eight and up. Hawaii High is more interactive story than adventure game, in keeping with the theory that girls like character and story more than fast action. It stars a teenager named Jennifer, who just moved from New York to Hawaii, where her mom, a working professional, has taken a job. She and her new friend Malaya are on an adventure to return a lost Tiki statue to its rightful place. Hot on their trail are two dastardly villains (a man and a woman—equally bad) who want the Tiki for its resale value on the black market.

Trina Robbins, who wrote and designed the game, said she used the Nancy Drew mysteries for inspiration. Those stories, said Robbins contain the classic things many girls like. "Girls haven't changed; girls will never change.

We will always like adventure and problem-solving." Robbins says she created two female protagonists because girls like to play together. She also wanted characters that girls could relate to and who were a little older than the game's target market. "Nancy Drew was a teenager but it was much younger girls who read her books."

An assumption game developers make is that girls like "fantasy," yet research states they like themes based in reality. Hawaii High takes both into account. Robbins uses elements of fantasy and mythology in the game, but the story takes place in today's modern world—and Robbins provides real-life situations her players can relate to. For example, we begin the game on Jennifer's first day of school, where she faces a classroom full of beachniks in shorts. Dressed in New York City urban chic, it's apparent that she feels like an outcast, as anyone would in the same situation.

Like many interactive stories, Hawaii High lets the players move at their own pace and interact with the environment. It also includes a "story map" that lets the player begin where she left off or "lost" the game. The things that work least well in the game, according to Robbins, are a few puzzles, which were thrown in by male programmers on her team to "make the game more exciting." In one, the player must maneuver the characters through a three-dimensional maze—an annoying interruption for any player, especially if she or he is trying to get to the next clue. Robbins says her gut feeling was that girls wouldn't like the puzzles because "they're closer to boys' twitch games."

Hawaii High's dress-up segment was criticized in the press, as was the choice of pink attire for one of the characters. Robbins calls such criticism "throwing the baby out with the bath water" in an attempt to avoid sexism. "Girls *do* wear pink, and I felt that it was one of the personality traits of this particular girl to wear lots of pink. The other girl character wears bright reds and blues." As for the dress up game: "Girls like to play dress up. Women do, too!" says Robbins.

Women and girl protagonists, especially in preteen adventures, are also prone to romance. Robbins purposely kept this to a minimum. "It's really the *idea* of romance that they like. At that age, most girls think little boys are horrible. Girls are much more interested in interaction between females." She took care of the romance issue with a scene in which Jennifer watches a cute boy play the ukulele and sing, while little hearts dance over her head. "Requisite romance taken care of," explains Robbins, "back to the girls!"

REFERENCES

The gender differences cited in this article come from a wide range of sources, including published academic and industry research as well as unpublished surveys. If you'd like more information on gender differences, here are some excellent sources:

- **"Gender and the Art of Designing Interactive Media," by Heidi Dangelmaier, *Computer Game Developer's Report*, Aug. 1995**
- **"We Have Never Forgetful Flowers in Our Garden: Girls' Responses to Electronic Games," Maria Klawe, et. al.; University of British Columbia, Dept. of Computer Science, Dec. 1993.**
- **"Exploring Common Conceptions About Boys and Electronic Games", Maria Klawe, et. al., University of British Columbia Department of Computer Science, Jan. 1994.**
- **"How the other Half Plays," by Barbara Lanza, *Computer Game Developers Conference Proceedings*, 1994.**

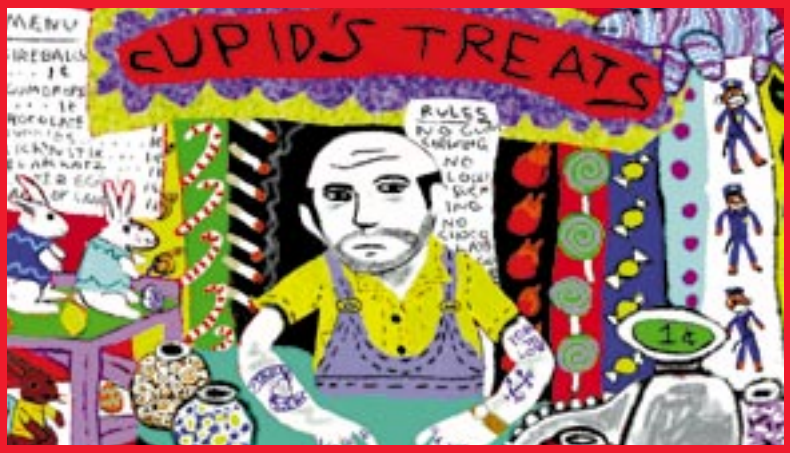
Chop Suey

Created by writer Theresa Duncan and artist Monica Gesue with the idea of creating a CD-ROM girls would like (but not a "game for girls," per se), Chop Suey from Magnet Interactive and 20th Century Fox isn't easy to define. It's neither game (no action, no rules, no scoring) nor interactive story (there's no beginning, middle, or end), nor creative activity program. You might call it an "interactive poem." The CD-ROM combines hip, engaging art, with melodious, alliterative prose read by National Public Radio's David Sedaris (who was selected for the job, according to Magnet, for his androgynous voice). In fact, if you were to give it edutainment value, it would be that it shows the beauty and power of words.

Duncan said she modeled the game after classic children's literature such as Maurice Sendak's *Where the Wild Things Are*. She said she didn't pay much attention to research, but "designed it from the heart." Chop Suey features two sisters: Lily and June Bugg, who live in Cortland, Ohio. In the first scene, all we see of them are their legs and anklette socked feet—one sister's legs are brown, the other's are white. Other than that, we see little of them throughout the game, since we experience everything from their point of view. The two girls have just gone shopping for red licorice at a candy store called Cupid's Treats and eaten chop suey with their father at the Ping Ping Palace. They're lounging in the grass, watching clouds pass overhead and trying to make sense of their shapes.

Soon, we're floating above a colorful map of Cortland, and the city is our oyster. We can explore the Carnival, Aunt Vera's House, Cupid's Treats, and several other spots on the map. Once we venture forth, we can return to the map with a click of the mouse—we're not going to get caught in a puzzle or a maze where we can't escape.

The "story" comprises memories and moments in a few character's past and present lives. We're voyeurs sneaking into bedrooms, peaking through windows, and rummaging through drawers. In Aunt Vera's room, her life unfolds as June Bugg and Lily remember things she's said before, embellish her stories, and fantasize about her adventures. (And none of these fantasies involves castles or witches, by the way.) We discover Miss Vera's life is "a long series of magical stories, like shiny, dull pearls on a long, long, necklace." Vera is a former Rockette who lived in New York City. Click on a picture on the wall and learn about her three ex-husbands (all named Bob). One Bob lives on the edge of town and still sends her "roses of the palest pink." Click on the matchbook cover on her bureau and go to a nightclub



Neither game nor puzzle, but a quiet exploration of a small town from the point of view of two sisters, Chop Suey's main focus is on interactive video, audio, and storytelling.

called the Single Spotlight, where a lounge singer performs a sad love song.

Even better, you can snoop in a boy's room. In this case it's Vera's teenage son, Dooner, a rock and roll musician with a motorcycle and a silk aviator scarf, who "used to sing songs about girls named Lurlalou, or Pitapat, or just plain Mary." Dooner sits on his bed amidst record album covers and bags of "Fame-On!" chips, oblivious that you're rifling through his bedroom drawers. We find a magazine called *Girly* (which we can't read) and his diary (which we can read), revealing an account of his date at the carnival with a painter named Monica.

In this game, at the picnic scene (where you watch Aunt Vera dance with an Elvis Presley look-alike named Ned), a lawnmower buzzes in the background, birds sing, wind chimes jingle in the breeze, and a dog barks. (If you pick up x-ray sunglasses at the bottom of the screen, you'll see Vera and her current beau in their underwear!)

For those who enjoy dress-up games, you have a few options to choose from. You can raid Aunt Vera's closet. Click on a long, red evening gown and see Aunt Vera lounging languidly in a hammock, a bowl of fruit on her lap. You can also dress up a drooling rambunctious dog named Mud Pup in loud boxer shorts, socks, and a shirt, while punk rock bounces in the background. In another segment, you put eyelashes, sunglasses, and a goatee on a woman's face.

Chop Suey isn't based on scoring or fast action, but on a mood, style, and sense of humor that kids and adults can enjoy. While the gender stereotyped experiences of romance and dress-up might sound corny, these elements are handled in a humorous, fun way. It's clear that Chop Suey appeals to a different sense of play—one based on exploration instead of target shooting and violence, one that delivers entertainment value from humor, characters, and the beauty and power of language and art. This defies the traditional "game" experience that boys seem to like so much, but is this type of computer play something that only girls can appreciate?

Elroy Goes Bugzerk

Elroy Goes Bugzerk from Headbone Interactive is another story-based game that boys might enjoy as well as girls—if not more so. The CD-ROM is billed as a "comic adventure for kids aged 7 to 97" and features a boy named Elroy and his dog, Blue, who are hunting for a rare stink-bomb-dropping beetle called the Technoloptera, in the hopes of winning their city's annual Insectathon.

This game's protagonist is all boy. He loves scatological humor (in one scene he enters a cave of locusts and comes out covered in locust poop). He's loud, he's sarcastic—but he's vulnerable, too. It's clear that Elroy is in a predicament: if he doesn't catch the beetle by the end of the weekend, he's going to lose the Insectathon and his nemesis, the evil Gordon Smugs, will win. It's your job as the player to help him.

The game features long stretches of animated action with few chances to use the mouse. Eventually, the story stops, Elroy asks you what he should do, and you tell him by clicking your mouse. Your actions usually involve telling Elroy to go one direction or another, exploring elements in a room, or choosing from a list of options, but most of them require some thinking. You need to use what you learn throughout the game (such as information about insects) to determine Elroy's—and the wisecracking technoloptera's—fate.

Like Chop Suey, Elroy relies on story and character as well as its own style of cornball humor. These elements lean toward stereotypic "male" preferences, which might turn off some girls. But the style of play lends itself to a wide range of characters and story elements—just as Chop Suey does. Adding a female heroine to Elroy's story could make the game more appealing to girls, especially if she's an important character who girls can identify with—and not just a "token" showpiece. According to Headbone, Elroy will get a female sidekick in the next title.

Appealing to the huge range of play preferences that girls and boys have is a great challenge for game developers, and one that might seem daunting. However, as interactive entertainment designer Heidi Dangelmaier puts it, "Gender is not a burden to bear but an intriguing human dynamic that can bring texture and passion to interactivity." To reach girls and women with electronic entertainment (and a new audience of boys and men), it's obvious that game developers need to consider new directions in computer play. Addressing different preferences and styles in gameplay (be they gender influenced or not) will make those new directions more interesting and exciting—for male and female gamers alike. ■

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

Void Pirates on Parade

David Sieks

Developing a visually exciting game takes more than the right design tools. Hard work, capital, and actors who'll work for beer are all part of the mix.

Digital entertainment has come a long way since Pong was able to captivate us for hours on end. As the technology has grown, so has the scope of game development. The digitized faces of big-name stars have taken the place of humble sprites, storylines are assembled by herds of suits, and graphics and programming are farmed out to production houses in faraway countries. In such big-stakes projects, Creative Control is an elusive bird: once the cage door is opened, off it flutters...and it doesn't come back when you whistle.

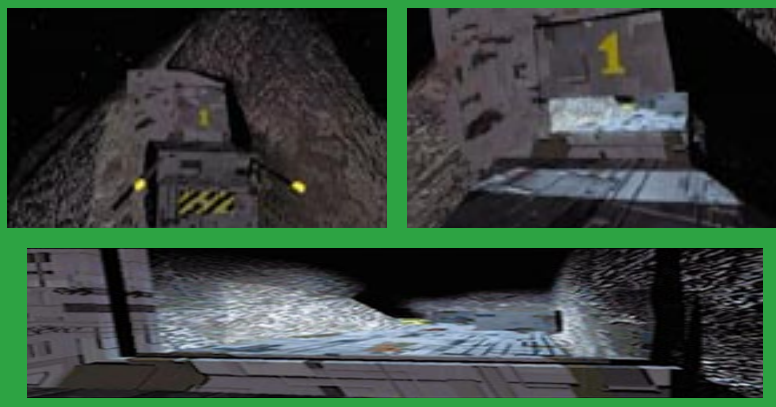
But if you're starting to think "I Did It My Way" is a tune no longer heard in the game industry, you should listen more closely to that humming sound emanating from Ian Firth. Firth is putting the finishing touches on Void Pirates from Diversions Software and SofSource Inc. I recently had the chance to ask him some questions about the project, and because I like you I'll let you in on what he had to say:

GD: Give us a little background: what is Diversions Software, what's your role there, and what did you have to do with the upcoming Void Pirates?

IF: Diversions Software was formed in 1992, after I created the artwork for WinFish, a joint venture of 2 Guys Software (myself and Andrew Novotak). I am based in Denver and am the sole employee. My role is "lone wolf," and I am responsible for every aspect of Void: code, art, animation, sound, even ad copy and box design.

GD: So what sort of projects has Diversions developed?

IF: I currently have one retail product on the market—Grey Wolf, a WWII U-Boat shooter—and a dozen shareware games and apps, including Prairie Dog



New features in Caligari trueSpace2—such as three-dimensional booleans and three-dimensionally rendered solid modeling—helped in the creation of this space station built into an asteroid, which players see from the main viewer on the bridge of their ship.

Hunt 2-Judgment Day, second place winner in last year's Ziff-Davis Shareware Awards. Two years of shareware development have included Prairie Dog Hunt for Windows and its sequel, Trap Shooting, Fortress, TailGunner, StarGunner, and a few applications including StarTex, a star texture generator for rendering programs.

GD: What's the connection with SofSource?

IF: SofSource licensed rights to a few of my shareware games and, after I sent them a demo of Grey Wolf in mid-94, agreed to feed me while I developed that project. I was recently out of work as a database programmer and jumped at the chance. I had always wondered why games cost so much to develop, so I set out to create Grey Wolf for $5K and did. Void came next, with a little bigger budget.

GD: How do you come to be making computer games? Are you an artist who's gotten into computers, or a techie type who's gotten into digital art, or someone who's always wanted to make computer games and just took on all the necessary roles, or what?

IF: I think more the techie type. The last decade of my life has been spent as a mechanical designer. I have been interested in graphics and 3D work since 1985, when first exposed to it. I spent time as a kid attempting to write games for the Atari 800 in Basic, but got tired of it after getting a car. My passion went from writing silly shooters for Windows to full-bore game development while creating Grey Wolf.

GD: What was that first exposure to 3D you just mentioned, and what about it interested you and inspired you to get involved?

IF: My first exposure to 3D was on an Applicon CAD CAM system at StorageTek here in Colorado. From there I moved to AutoCAD. The first animation I did was a flyby of Durango airport with AutoCAD 9 in 1989. Just seeing what was possible is what drew me into it, and I don't think it will ever end.

GD: There's a lot of rendered 3D in Void Pirates, which I understand was done with Caligari trueSpace. With your CAD background, how did you settle on that tool?

IF: My background in 3D includes 3D Studio, Macromodel, AutoCAD AME, and anything else I could get my hands on. trueSpace was chosen due to price when Grey Wolf began, and I quickly adapted to it. Money was not available for 3D Studio and, even if it were, I would not have used it due to the fact that it is DOS based.

Moving beyond the single-screen user interface, Void Pirates uses a main bridge screen with seven hotspots leading to detailed control screens.

When Void Pirates started, I picked up a second system (P90), for rendering, and stuck with trueSpace. The only limitation was lack of 3D Boolean operations, but I still managed. When trueSpace2 showed up, I did go back and redesign some of the ships in the game using the new Boolean features. I also created some interesting stations built into tunnels on asteroids, which couldn't be done in trueSpace 1.0. I just wish the motion blur option was faster.

GD: What made you stick with trueSpace after Grey Wolf?

IF: trueSpace is the easiest, most refined piece of software I've ever seen. There is so much packed into 1.2 MB of EXE: no DLLs, nothing. The interface is also the finest in the world, making designing so much easier. Win95 uses cascading menus, something I have hated since AutoCAD 2.6. I can't believe Microsoft would go back so far in user interface design. Another reason for using trueSpace is the plethora of game companies touting "3D Studio-Rendered Everything!!!" I want to be able to say "Not a single thing was rendered in 3DS, and not a single actor you will recognize!" My actors are all people from a bar downtown and are working for beer.

Money is another reason. I don't have $4K plus another $5K for plug-ins for 3D Studio. I paid $469 for trueSpace...and I've seen some terrible 3D Studio stuff hit the shelves.

I am shipping the trueSpace2 demo along with some models with Void Pirates.

GD: What other tools did you make use of to create the graphics for Void Pirates?

IF: Everything was done on a Digital P90 with 40MB of RAM, 2GB [of storage], and Stealth 64 VRAM. Post production was done with Adobe Premiere and in:sync RAZOR. All textures (400MB worth) and touch-up were done with Corel Photo-Paint 5+. Live footage was shot with a Sony Hi-8 against my living room wall. Footage was captured with an Intel ISVR Pro card. All artwork in the game uses the Indeo 3.2 fixed palette. There is a bit of graininess because of this, but palette problems are hexed. There is a little choppiness to some of the images, but I am willing to sacrifice image quality in favor of easy, flash-free palettes.

GD: Let's talk about the look of the game. The sci-fi setting in Void Pirates is realistic and nicely detailed. What were some influences, and what did you try to do to differentiate your game visuals from sci-fi imagery we've all seen before?

IF: I was always impressed by the style of the original Aliens movie. The interior of the Nostromo was very well done—people smoking, porno magazines, empty coffee cups, and the two bobbing ducks in the main area really gave a realistic look to life in space.

The Orinoco (the player's ship) was designed after looking at the mess on my desks here at home: ashtrays, soda cans, general disarray everywhere, but I still know exactly where everything is. Since space is dark, the overall mood of the game is dark; a sad future similar to *Blade Runner*. The shininess and lighting of the *Star Trek* series always seemed too happy. Void Pirates focuses on trading of narcotics, stolen property, and the like, and a darkly lit design sets the mood (hopefully). The darkness also helps by masking areas where there should be more detail but I couldn't afford to put it in.

GD: Couldn't afford in what sense?

IF: Total budget was around $20K—this includes my rent and bills. Seven months, $20K, and one person is very little to create a saleable multimedia product these days. I would love to spend twelve months on Void Pirates and $100K, but it isn't available, so I have to create what I can with what I have in as little time as possible. The design was limited by what I could render with 40MB RAM, without hitting the swap file. Time is limited developing a game with one or two systems. Total hours for modeling and rendering would be about 800 so far.

GD: So far? What remains to be done?

IF: More animations, more cut scenes. The still artwork is 99% finished. I think of something new each day. I'll spend thirty minutes trying to get an idea into trueSpace2. If I can't, I give up and try something else the next day.

GD: Will the game feature a lot of animation?

IF: The majority of the animations are actual gameplay. Void Pirates uses many concepts from other games. Ships are attacked during strafing runs similar to Rebel Assault, displayed with an AVI. After disabling a ship, the hull is breached with a drone, which you then pilot around the interior of the ship as in Iron Helix. The animations are not the main point of the game, and I have tried to integrate them as seamlessly as possible. There are also cut scenes that play as the player progresses through the game.

GD: You voiced some strong opinions about user interface design earlier. One of the great logistical and creative challenges in game design is creating an interface that manages tos atisfy all the various gameplay requirements, coexist peacefully with all the technical limitations of the minimum target system, be readily usable by the player, and stillf it the game milieu and look good.

How did you approach this task in Void Pirates?

IF: The Void Pirates user interface has changed only a little during development. I had grown tired of the single-screen user interface (such as Daedalus Encounter, Journeyman, Iron Helix) and wanted the player to see more. Grey Wolf has a Myst-type interface, where the player can walk through the ship. The only problem is a lack of crew members. It was basically a ghost ship, and the player had to do everything.

In Void Pirates, the user interface consists of a main bridge screen with seven hotspots. Five of them take you to closeups of the control clicked on, and the other two take you to the Main Cannon or the Gun Turret. Originally, a walkthrough design was used, but the image quality suffered from compression, so still image interfaces it was. The player can get tired of walking around the ship in short time—as in 7th Guest—so stills make sense.

To lose the ghost ship image, there is a VidPhone the player can use to talk to crew members. The crew members are live video shot against blue screen here in my home. All communications with nonplayer characters are handled via remote video drones or a text based terminal. All selling and trading is handled in this manner, and the player never leaves the ship while in port.

System requirements are MPC2 for a minimum, DX2/66 8MB RAM recommended, accelerated video highly recommended. I expect a lot of people calling tech support saying their Packard Bell 486/25 won't play the game. I've run into about 90% of Grey Wolf customers having their systems set up wrong for decent Windows performance. The only limiting factor in the game is video card speed, and CD-ROM transfer speed.

I did run into a problem with CD-ROM playback speed. My limit of 270kps AVIs has caused a bit of image degradation, and there are certain parts of the game where frames cannot be skipped. There is a graphics diagnostic program, along with VidTest in case the user needs to determine whether their system is MPC2 compliant.

GD: So what's next after you burn in Void Pirates?

IF: I and two other developers are coming out with a Visual Basic Game Toolkit, which will include a lot of source from Void, plus a replacement for SurroundVideo (scrollable, scaleable virtual reality AVI files) I have come up with using Visual Basic and trueSpace. Hopefully the Toolkit will get finished this year. It is in the planning stage right now, and I have very little time to work on it. It will include source for various types of games: scrolling, platform, shooters, AVI walkthroughs... Plenty of code for blitted buttons, instead of the standard Windows look. The other two authors are contributing sound and music goodies, and business multimedia stuff, as it isn't aimed at game design only.

If we can find some capital, Andrew (WinFish guy) and I are going to hook up again. He is working diligently with the Intel 3DR toolkit, and we are thinking along the lines of a networkable Tank Simulation, due to the speed of 3DR. With trueSpace, we should be able to create a very nice interface for a Tank.

And yesterday, SofSource and I agreed on a vastly improved Grey Wolf 2, for next spring. At least I don't have to wear a tie for another eight months. ■