



GAME DEVELOPER MAGAZINE

OCTOBER 1997



More Caviar?

I read Dave Sieks' interesting report about Caviar technology ("Dawn of the 3D Pixel Sprite," pp.58-61, *Game Developer*, June 1997). I am a programmer with a restricted budget. I want to experiment with something like this technology. I guess it's Animatek's secret, but would you guide me further? How could I get more information about the technique? If you know of some literature references it would stimulate my imagination.

Ahmet Engin Karahan

CONTRIBUTING EDITOR DAVE SIEKS

RESPONDS: Yes, Animatek has patented their Caviar technology. As mentioned in the article, you can download freeware Caviar viewers from their web site (<http://www.animatek.com/VoxelCh.htm>), which will at least afford you a glimpse of the process in action. Also available for licensing by game developers is another implementation of voxel sprite technology from Attention To Detail. You can find details and a downloadable demo online (<http://www.atd.co.uk/voxel.html>). And though it's not voxel sprites, you might be interested in checking out a list of 3D voxel landscape engines maintained by Karsten Isakovic at the Technical University of Berlin. The list currently boasts 33 different engines, with available source code and documentation - (http://cg.cs.tu-berlin.de/~ki/3de_land.html). Good luck.

Algorithms for Warcraft, C&C

That was a great article on multiple object pathfinding in the June 1997 *Game Developer* ("Real-Time Pathfinding for Multiple Objects," pp 36-44). What type of pathfinding algorithms do popular real-time strategy games such as WARCRAFT II and COMMAND & CONQUER: RED ALERT use when moving large numbers of vehicles? It's great to learn how to do things the right way, but as a game developer, I'm also interested in learning how the other guy did it.

Dylan Miller

AUTHOR SWEN VINCKE RESPONDS: I have to warn you, this is pure speculation, and I'm doing this from memory. WARCRAFT II

probably uses a variation of A* with a limited search horizon, possibly using something similar to the cut-locked path method for avoiding other units. I guess they also had some look-up tables in there to decide which paths were impossible (the way their sea transport boarding algorithm works suggests this). While I think the original COMMAND & CONQUER used a variation of the wall-tracing algorithm, RED ALERT probably uses A*. Unit avoidance there is almost definitely step-based (the way the units deal with blocked bridges points in that direction), but some interunit communication might also be present. Obviously, RED ALERT also has code in it that allows several units to follow a main path, which is easy to do if you work with step-based unit collision avoidance.

Rant

I would like a moment of your time to let you know that I like the "new" look, but I am a little fearful that the content will drop to advertising like other "used-to-be-good" magazines.

I own every issue of *Game Developer*. I enjoy it that much. The problem is that I have already noticed a drop in content; *Game Developer* spends more and more time with "talk-to-this person" stuff, "political" issues (in the form of "what I think about the industry") and "advertising" in the form of new game explanations.

Yes, I like to know when the games come out and if they're worth looking into, but I don't want to see the important things such as how-tos and examples disappear to this "talk-mag" style (examples past and present include *Commodore* and *Byte* magazines).

I am sorry that I felt compelled to write a not-so-positive review, but I just don't want to lose the valuable information I glean from articles in *Game Developer*. Thank you for listening.

Gregory L. Miller



Hey Folks! Drop us a line at gdmag@mfi.com. Or write to *Game Developer*, 600 Harrison, San Francisco, CA 94107.

with Freeman's thoughts on how to allow a game design document to guide the development without restricting the creativity and allowing "sparks" of brilliance to improve the overall product when appropriate.

Anyway, I just wanted to say thanks. Keep up the good work!

Chris Longpre

Design Documents

I just finished Tzvi Freeman's article ("Creating A Great Design Document," pp.58-66, *Game Developer*, August 1997). Many thanks! The article provided many insights for me and several ways that I can improve my next game development cycle.

I am a producer in the game industry, and my last project had me writing the design document. Many of the foul-ups that can happen, did (it gave me a good chuckle to read that they happen to others too!). I especially liked your acknowledgment of the "soul" of a game, and how it can focus those wildly creative members

of the team. I also agree with Freeman's thoughts on how to allow a game design document to guide the development without restricting the creativity and allowing "sparks" of brilliance to improve the overall product when appropriate.

Anyway, I just wanted to say thanks. Keep up the good work!

Chris Longpre

Secure Them Pants!

Your editorial in the July 1997 issue of *Game Developer* made me laugh out loud when I visualized "security breeches." Yeah, I guess that confirms that I'm a geek.

Anyway, I know that you know what you meant.

Dan Kollmorgen

EDITOR ALEX DUNNE RESPONDS: That is really funny. You're the first person to point that out to us, and I nearly died laughing when I reread that particular sentence. Therein lies the main problem with spell checkers — there's no way to check the context of the sentence with them. Anyway, perhaps security breeches are a good idea too, in certain situations!

The Zenith of Hardware Accelerated Rasterization and Beyond...

Last month, I talked about what I'd like to see as the baseline of features for 3D accelerators in the 1998 timeframe. This month, I'm going to get into the future of 3D acceleration, including the apex of hardware accelerated rasterization and moving beyond the triangle model of representing and rendering data.

10

The Peak of Rasterization

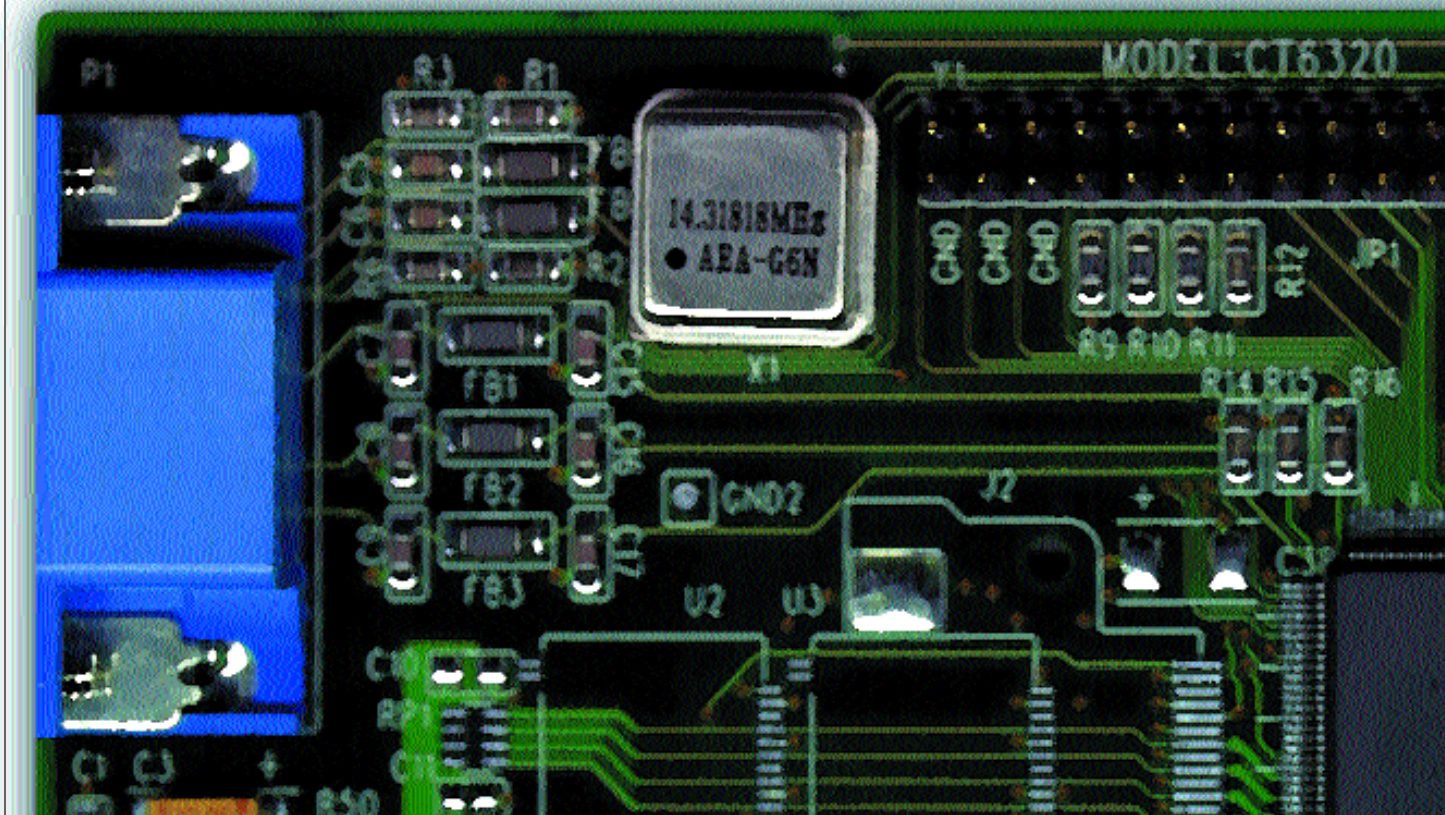
For the next few years, we're still going to be using hardware accelerators that represent objects as triangles. This model is still viable for a few more years, but it's getting a little long in the tooth for a lot of applications, and with games pushing the envelope far harder than "real" applications, I wouldn't be surprised to see games that represent data in radical new ways.

So I'd like to finish up our coverage of hardware accelerators and then talk a bit about what the future of hardware acceleration and 3D graphics might look like when it comes to games.

24-BIT AND 32-BIT FRAME BUFFERS. Twenty-four-bit (RGB) and thirty-two bit (RGBA) frame buffers will become increasingly important in the future as fill rates increase and developers use that fill rate for multipass rendering techniques. As I've bemoaned before, multipass rendering can introduce

some noticeable quantization error with 16-bit frame buffers. Significant errors can be introduced as early as the third pass. With 24-bit frame buffers, however, quantization errors due to multipass rendering are greatly reduced.

Currently, hardware vendors are shying away from deep frame buffers because of their higher bandwidth requirements (deeper frame buffers consume more refresh time and also require more bandwidth for fill operations) and their greater memory foot-



print. But with memory speeds getting faster and memory parts getting cheaper, I'm hoping that deeper frame buffers become a reality.

A 32-bit frame buffer with a destination alpha channel would be even better — if you give game developers more buffers, they will figure out something cool to do with them (such as alpha saturate antialiasing).

PER-PIXEL MIP MAPPING AND BETTER FILTERING.

As of today, the only accelerators that I've seen with usable per-pixel MIP mapping have been the PowerVR PCX2 and the 3Dfx Interactive Voodoo. I used to think it wasn't a vital feature, but after seeing GLQUAKE's horrible aliasing artifacts on non-MIP-mapping hardware, I've revised my opinion. Per-pixel MIP mapping really is going to be a necessity.

While I don't consider trilinear filtering completely necessary, anyone who has played GLQUAKE on a 3Dfx Voodoo-based graphics adapter has probably seen the noticeable MIP-map banding effect when running down a hallway. Trilinear filtering is a memory bandwidth hog since it requires twice as much data to be fetched as bilinear filtering. Still, trilinear filtering is definite-

ly useful, especially if there is no performance penalty for enabling it.

When it comes to filtering, I guess it boils down to this: the less aliasing and visual artifacts, the better. If a hardware

Currently, hardware vendors are **shying away from deep frame buffers** because of **their higher bandwidth requirements and greater memory footprint.**

accelerator does this by traditional trilinear-filtered MIP mapping, anisotropic filtering, or by contracting the antialiasing fairy to sprinkle magic filter dust on every chip that gets stuck on a board, I don't care — just make texture aliasing go away, hopefully without blurring everything into the Hell of Marshmallowy Pixels at the same time.

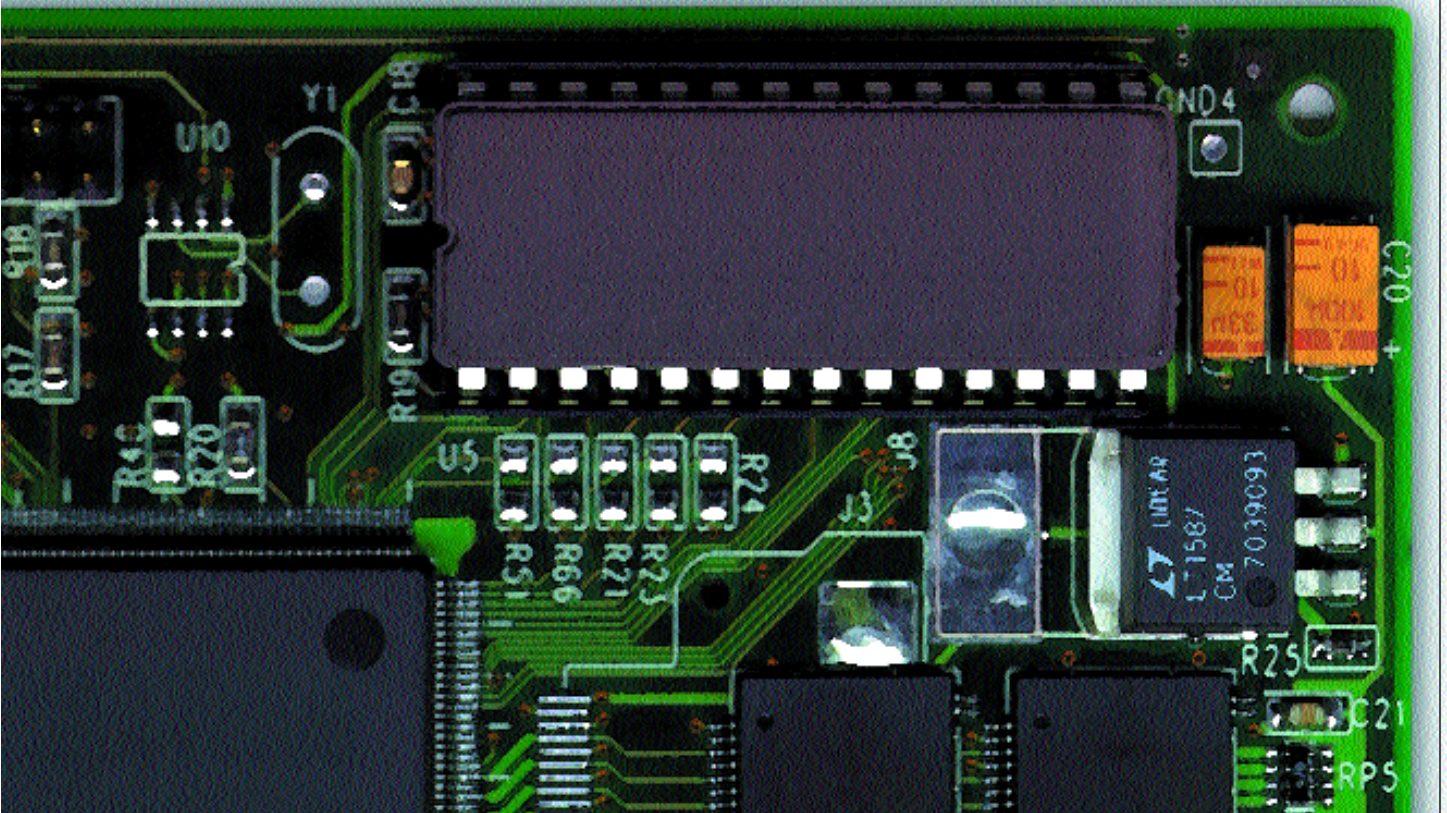
24-BIT AND 32-BIT TEXTURE MAPS. We'll be reaching the limits of what can be accomplished with 16-bit artwork pretty soon (especially the 4444 ARGB format, where 4 bits simply isn't enough for each channel), and game developers will want the ability to selectively fall

back to very high resolution, 8-bit per component texture maps.

24-BIT AND 32-BIT Z-BUFFERS. The limitations of 16-bit Z-buffers are becoming obvious even with today's games —

GLQUAKE exhibits some Z-buffer aliasing artifacts on 3Dfx Interactive's Voodoo accelerator's 16-bit Z-buffer. A move to a 24-bit Z-buffer format would help with Z-aliasing problems greatly. A 32-bit Z-buffer might be overkill, however, but I imagine in a few years it will be the norm.

UNRESTRICTED EDGE ANTIALIASING. Unrestricted edge antialiasing has been talked about for a very long time — improving image quality without requiring larger resolutions, more memory, or inconvenient programming techniques is one of the Holy Grails of computer graphics. The ability to have



antialiased polygon edges without any significant application burdens is probably going to be one of the key technologies to arrive in the next generation of consumer accelerators (assuming someone doesn't figure it out, patent it, and screw the rest of the industry). Today, you can do a hacked-up form of antialiasing using alpha saturate techniques with some accelerators, but it requires rendering in front-to-back order.

A nice side benefit of computing data on the fly is that level-of-detail can be controlled very precisely, letting future software scale very cleanly from very fast to slow machines.

Complete scene sorting is no longer acceptable for modern games, especially if all it buys you is antialiasing with a select few accelerators. Developers are adopting the Z-buffer for hidden surface removal in droves, so the likelihood of future games keeping data in an easy to sort format is very slim. Even ignoring the data structure issue, rendering in depth-order introduces the problem of excessive state changes, which won't make things any faster.

STENCIL PLANES. Stencil buffers have a lot of interesting uses, including applications that use mirrors, shadows, and reflections. Once again, this comes down to the same philosophy – build it, and we'll find a use for it.

MULTIPLE TEXTURE SUPPORT. Wouldn't it be nice if we could do multipass rendering in a single pass? Well, this is the crux of multiple texture support — the ability to specify multiple textures and texture coordinates for a single triangle. Future hardware should support this technique, as it can scale performance nearly linearly with the number of passes saved.

INEXPENSIVE STATE CHANGES. State changes are expensive with today's hardware. Because of this, game developers are having to bend over backward trying to minimize state changes as much as possible — sorting polygons in material order, rendering similar objects at the same time, and so on.

Ideally, rendering order shouldn't

be dictated by hardware designs, so it would be nice if any state changes were done as quickly as possible, especially things like changing the texture environment, the light mode, and the current texture. Changing the current texture should not take a lot of time.

PERSPECTIVE-CORRECT LIGHTING ITERATORS. This actually addresses two separate issues. For starters, it would be nice if hardware supported independent diffuse and specular lighting iterators —

Direct3D exposes this already, and some type of implementation of this will hopefully be introduced into OpenGL at some point in the near future. Game developers can always find uses for more iterators.

Also, it would be nice to start seeing perspective-correct lighting iterators. For the longest time, developers assumed that perspective correction for Gouraud shading wasn't necessary. In reality, this hasn't proven to be the case – sure, with very high polygon count scenes where your average triangle size is four pixels, perspective correct lighting (and even texturing!) isn't needed. But for a lot of game titles where you have large walls and floors, linear lighting just looks weird.

BETTER FOG. Both Direct3D and OpenGL are pretty bad when it comes to specifying fog — you can specify the general types of curves you want (linear, exponential, or exponential squared), but you can't actually specify an arbitrary set of fog densities or a general equation for fog. With some APIs and hardware, such as Glide for the 3Dfx Interactive Voodoo, you can actually upload a fog table yourself. This is really handy for achieving cool effects other than fog. You can do a "free" blend to a constant color à la VGA palette tricks without having to do a screen-sized alpha-blended polygon.

Unfortunately, coming up with a clean abstraction that isn't fundamentally broken on everyone's hardware is

tough, so this is sort of wishful thinking. Nonetheless, if OpenGL and/or Direct3D expose a more flexible hardware abstraction for fog, I'd really like to see hardware vendors support it.

But this presumes that the "fog this pixel based on some function dependent on screen Z" model is the right direction, and I'm unconvinced that it is. John Carmack has convinced me that the generic fog most accelerators implement is generally unimpressive and boring. Since I came from a hardware accelerator background, I've always thought that hardware fog was neat, but that was more of a historical bias than an honest opinion. When you think about it, hazing stuff based on screen Z is just... cheesy. There's no real sense of realism there — there's a big difference between a static hazing effect and multicolored patches of fog swirling in and out of windows.

Next-generation games want to implement true patchy fog, swirling mists, streamers of smoke, all integrated effectively with each other. A cheap distance attenuation effect doesn't give us these effects. In order for us to achieve this kind of control, we need to be able to have a fine grain control over fog at each pixel, or at least at each vertex if triangles are small enough. Last month, I talked a little about fog color iterators, and this is definitely the direction I think things need to be heading for triangle-based hardware accelerators.

HOW MUCH RAM IN OUR ÜBER-ACCELERATOR?

Okay, I know I'm asking for a lot, and I don't necessarily see even a fraction of my wish list being implemented (unless RAM prices manage to drop yet lower, which I find hard to imagine), but let's just see what kind of RAM requirements we're asking for from the Ultimate 3D Accelerator. Let's assume a 32-bit ARGB double-buffered frame buffer, a 24-bit Z-buffer, an 8-bit stencil plane, and 4MB of local texture RAM (which may not even be necessary if AGP works out, but I'm not holding my breath). Depending on the screen resolution, we're looking at almost 8MB to run at 640x480, and around 20MB to run at 1280x1024. This presumes we're running in a full-screen exclusive mode and not sharing memory with the Windows desktop — things get worse if we're trying to run in a window.



Now, a consumer-level 8MB 3D accelerator isn't that far-fetched, especially given that several 8MB Windows accelerators are available even today based on 3Dlabs and Number Nine chipsets (8MB is necessary to support a Windows desktop of 1600x1200x32-bits). Today, there are high-end accelerators that can come with 16 or more megabytes of RAM, and prices for graphics adapters have been plummeting the past few years. The price of a 4MB Windows accelerators today is between one-half and one-fourth the price of a comparable accelerator circa 1995.

So while it's theoretically possible that we'll see 12MB and 16MB accelerators coming in at the consumer level in a couple years, I don't expect to see this anytime soon. The fact of the matter is that you need less than 11MB to run an 1800x1440x32-bit Windows desktop, and very few Windows users need that much real estate (you'd need a big monitor for that to do anything but cause eyestrain). Our fully loaded accelerator would get by with 8MB at 640x480, which is a pretty reasonable resolution to work with, especially if antialiasing is available.

Anything past 8MB is simply gravy — higher resolutions and more texture RAM, and that's it. And if that's all the extra RAM gives you, it may be a pretty tough sell. But hey, that's what they said about systems with 32MB of system RAM, so I'm probably wrong. Forgive me if my imagination isn't what it should be.

What's Wrong with the Triangle?

So far, I've described everything that we'd like to see from triangle-based hardware accelerators. However, I don't think the triangle model of rendering things is long for this world. It's been around for over a decade now on workstations, and its weaknesses and failings are becoming all too clear on the PC platform now that game developers have started thrashing on it.

So what's wrong with this model? Essentially, the triangle/polygon rendering paradigm requires specifying a rendering state (texture, lighting, Z-buffer functions, and so on) and then tossing polygonal primitives at the accelerator. Herein lies the rub — where do the polygonal primitives

come from? The source data must be computed or fetched, then it has to be transferred to the hardware accelerator across a bus, and finally the graphics accelerator has to digest and regurgitate the data in a pleasing form.

This model is rife with potential bottlenecks. If we're using a lot of precalculated data stored in main memory, then memory bandwidth could be a bottleneck. If a lot of complex data is being generated on the fly, then the CPU could be a bottleneck. If a lot of data is being sent over the bus, then the bus could be a bottleneck. And if our scene is particularly pixelicious (lots of big polygons, lots of overdraw, and many rendering passes) hardware rasterization performance could be the bottleneck. So we have all kinds of things waiting to thwart us in the performance department; in all likelihood, any problems will arise from a combination of these bottlenecks.

MEMORY BANDWIDTH. Main memory performance has been a problem since the introduction of the Pentium, and it continues to be a problem on very high performance processors. With this in mind, it's probably a bad idea to store lots of data in main memory. Thus, traditional static model representations (lists of vertices and connectivity data) aren't the right thing anymore.

CPU PERFORMANCE. Since CPU horsepower keeps growing at a phenomenal rate, it would be safe to assume that we can lean on the system processor to make up for memory bandwidth constraints. This means computing data dynamically from procedural representations whenever possible, rather than using stored data — a 180-degree turnabout from the days of the 386, when "tables, tables, and more tables" was the way of the day.

A nice side benefit of computing data on the fly is that level-of-detail can be controlled very precisely, letting future software scale very cleanly from very fast to slow machines. A level frame rate is maintained, but slower systems get coarser looking worlds and models. A reasonable enough trade off, especially compared to today's situation, where slow machines get slide shows.

BUS BANDWIDTH. Assuming that we want to generate scenes with lots of depth and geometric complexity, and conveniently ignoring the issues of gen-

erating the requisite data, we still need some way to get our data to the accelerator. Triangles can consume a lot of bus bandwidth. Assuming the unlikely case that our data is perfectly stripped, meaning that there is a ratio of a little over one vertex per triangle, and assuming that vertices consist of x, y, z, s, t, w , and two packed color values (à la Direct3D), then each vertex consumes 32 bytes. Say a game slated for release in 1999 requires one million triangles per second, then this game will consume 32MB/second of bus bandwidth just for vertex data. That's quite a bit, especially given that there is also going to be contention on the bus for state data, texture downloads, audio samples, and whatnot.

If the data is not stripped, then multiply that figure by three — 96 MB/second required just to render those triangles. PCI can barely do that today, assuming that the entire bus's bandwidth is devoted to triangle data; maybe AGP can solve some of these issues. Either way, cranking up bus bandwidth is only a stopgap solution. Clearly, then, the right thing to do is to reduce the amount of data being sent across the bus.

FILL RATE. Software can't do much to make up for poor fill rate. It looks as if the upcoming crop of 3D accelerators isn't going to be all that bad, and if we can count on 3D chip manufacturers to maintain their current pace, we should be O.K. when it comes to fill rate for the foreseeable future.

What's Better Than the Triangle?

So there are all these potential bottlenecks — memory bandwidth, CPU performance, bus bandwidth, and fill rate. There are different approaches to solving these problems, some of which are the Wrong Answer and some of which may be the Right Answer.

THE WRONG ANSWER. Some hardware vendors are using the brute force method right now to solve these problems, which often introduces new problems along the way. For example, chips that accelerate transformations and lighting have been announced. The advantage here is that the chips offload work from the CPU — but is this work really worth offloading? I don't know the



answer to that, but today I just don't see transformations and lighting being the key bottleneck.

To alleviate bus bandwidth problems, some hardware vendors are investigating caching vertices across the bus, a form of display list. For example, an application could define a set of vertices and faces and call it something such as "airplane," at

I can't fault Nvidia for developing the NV1— fundamentally, the concept was sound. It was the implementations and timing that really hurt its acceptance.

which point the hardware driver uploads the data (across the bus) into card memory. Now the "airplane" can be rendered by simply sending along a very small rendering command.

The problem with this way of doing things is that it presumes that we're going to be using static geometry when these hardware accelerators finally reach the market. As I've stated earlier, I find it highly doubtful that in the future games will have models made up of discrete points and faces. Future games will likely procedurally generate model data on the fly, so caching vertices across the bus is of dubious value for next-generation titles.

So what's the right answer? That's hard to say. It's clear that the triangle model of rendering isn't going to be the answer after a few more years. Something that doesn't play havoc with bus and system memory bandwidth will have to be developed, and in all likelihood, this may be some form of retained-mode hardware architecture (or an all new way of CPU-based rendering that is leaps and bounds better than what you see from a hardware accelerator). Something else we may see is hardware that isn't quite retained mode (as in, fully scene oriented), but instead takes simple descriptions of complex surfaces and handles the generation of the complex rendered data in hardware.

Nvidia developed something like this with their NV1 chip, which wasn't very well received by developers. I can't fault Nvidia for developing the NV1 — fundamentally, the con-

cept was sound. It was the implementation and timing that really hurt its acceptance. Developers weren't ready for patch-oriented rendering, and the NV1 didn't have the features or performance necessary to really push a new graphics paradigm.

NEC/VideoLogic has also attempted to utilize a non-triangle-based architecture in the PowerVR PCX1 and

PCX2 accelerators. The PCX1 wasn't very fast, and it lacked bilinear filtering; once again, developers weren't ready to adopt such a wildly new architecture (the so-called "infinite planes" method of scene and object description). The PCX2 addresses a lot of these issues, but convincing developers to support infinite planes directly isn't going to be easy, which means that PCX2 has to rely on good Direct3D drivers that convert triangles into associated sets of planes.

NV1, PowerVR, and even Microsoft's Talisman were and are pointing in the general direction we need to be heading. At some point, we're going to hit a wall in terms of transformation performance, bus and memory bandwidth, and fill rate, and the triangle paradigm isn't going to have the legs to keep going. What the new paradigm will be is unknown at this point, but maybe in a year or so we'll have a better idea of where things need to be going.

And on a related note, whatever new paradigm is adopted may not be particularly amenable to the existing immediate-mode APIs that we're using today (OpenGL and Direct3D). We may have yet another set of API wars in the next five to six years.

The Royal "We"

I've taken a lot of liberties with the pronoun "we," but a lot of the suggestions and features that I've outlined in this month's column have

come directly from game developers. If you're a game developer and think I've missed something, please let me know and I'll pass it along. If you're a hardware developer and think I'm smoking crack, please let me know and I'll tell you why you're wrong.

Spread the Knowledge

I'd like to point out that many of the techniques that I discuss were not invented by me. I'm simply your humble scribe, writing down these techniques. For example, much to my chagrin, I realized that I neglected to note the contributions of Gary McTaggart of 3Dfx Interactive in my August 1997 column on multipass rendering. Gary has done a lot of work for the game development community while at 3Dfx, by both exploring techniques such as multipass rendering and lightmap technology with radiosity, and, more importantly, exposing these techniques freely to game developers around the world (including presenting a very well-received talk at this year's Computer Game Developer's Conference). There are many people like Gary who are active in the game development community and contribute, but we can always use more people like him.

So I urge any of you with cool ideas to write articles, books, web pages, or Usenet articles discussing them. Simply put: spread the knowledge. What goes around comes around. ■

Brian Hook is a columnist for Game Developer who worked briefly for a hardware company and thus now foolishly thinks he's a chip designer. When not posing as an industry pundit and futurist, he spends his time at id software working on QUAKE 2. You can reach him via e-mail at bwh@wksoftware.com.

FOR FURTHER INFO

3Dfx Interactive

<http://www.3dfx.com/>

Nvidia

<http://www.nvidia.com/>

Number Nine

<http://www.nine.com/>

NEC/VideoLogic


<http://www.videologic.com/>



PRODUCING INTERACTIVE AUDIO: THOUGHTS, TOOLS, & TECHNIQUES

25

imes have changed in the game development industry.



All of the stupid money has fled to Internet commerce applications, and companies in the game world are left having to make their way as viable businesses. There is no market tolerance left for “junk,” or as it used to be referred to by a producer at Sega some years ago, “library titles.” The “B” and “C” titles that filled out your publishing profile aren’t viable any more — in fact, they can be downright deadly. Every game made from now on has to have a legitimate shot at becoming a hit; otherwise, it’s just not worth making. This philosophy has inevitably trickled down to audio departments, where the focus now is squarely on quality.

A Rant on Interactivity

What exactly does it mean to produce interactive audio? For some time, interactive audio suffered from an identity crisis. The term has come to mean less and less. In my own jaded way, I always imagined the adjective "interactive" modifying a noun such as "media" or "audio." When a person says that they are "doing interactive" in reference to audio, it usually means the individual

has stumbled into a job working on a CD-ROM or web site and is desperately trying to figure out how make 8-bit, 22Khz Red Book audio not sound as if it's being played back across two tin cans connected by string. To this person, "interactive audio" simply means sound for nontraditional media: CD-ROMs, console games, kiosks, and web sites. To me, the term means something entirely different.

If you're describing audio as "interactive," you're implying more than

just linear playback. Interactive audio should be constructed in such a way that the user can affect its performance in real time during playback. I'm talking about reactive, responsive audio, coming from audio drivers that are "aware" of what's happening and can respond by changing the music appropriately. Spooling a two-minute pop song in an unchanging, endless loop during a real-time strategy game is not interactive audio. Perhaps the term "audio for interactive media" would be appropriate instead.

Imagine instead audio that is an interwoven part of a 3D, free-roaming world. As you explore the world, the sound smoothly accompanies you, rising as you encounter danger, falling away as you explore in peace. It sounds triumphant when you succeed, and distant and mournful when you fail. And all of this happens with the precision and emotional impact of a great film score. In a user-driven world such as a game, you have no linear timeline by which to synchronize the changes in the music, as you do in a movie. The audio entirely depends upon the unpredictable input of the user. How can you make this work?

The answer lies in the nature of an interactive 3D world and is made possible by new tools and technologies. The 3D game world is open-ended, a database of terrain, objects, animations, behaviors, and their various relationships. Therefore, the music must also become a database of musical ideas, sounds, instruments, and relationships, imbued with awareness of the other objects in the world and programmed with responsive behaviors of its own.

The Rules of Interactive Sound Design

Over the years, I've worked on about 100 titles, 60 or so in a substantive way. I can distill much of what I have learned from this in a short set of rules.

1. There will always be limitations. Hardware limitations, space limitations, design limitations... you name it, and it will be restricted at one time or another. The only resource that's never limited is your

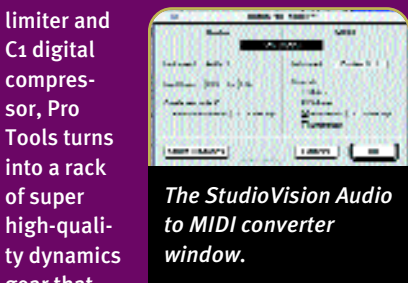
Tools of the Trade

Workstations and Sequencers

I try to use off-the-shelf tools whenever I can. When I evaluate commercial tools, I look for unique functionality, efficiency, reliability, and cost-effectiveness. Our main control room at Crystal Dynamics is based around a Macintosh Power PC running Digidesign's Pro Tools III 4.0, Digital Audio workstation hardware and software, and Opcode's Studio Vision Pro 3.5 MIDI plus digital audio sequencer. Pro Tools was an easy choice for me because of the TDM (time domain multiplexing) real-time plug-in architecture. This allows me to use Pro Tools as a general purpose DSP platform. In addition, version 4.0 is native PowerPC software and is much faster than the previous version. The upgrade includes extensive real-time automation, rivaling or surpassing that of most high-end mixing consoles. The value of Pro Tools as a platform really shines when you get into the plug-ins from K.S. Waves. With L1 digital



StudioVision Pro 3.5



The StudioVision Audio to MIDI converter window.

limiter and C1 digital compressor, Pro Tools turns into a rack of super high-quality dynamics gear that can be used to control unwieldy audio volume levels and minimize noise. These two plug-ins are excellent for maximizing dynamic range and are essential for anyone who needs to work in low bit rates or just needs to be loud. Wave's Q10 is the most flexible and best sounding equalizer that I have used, and their Truverb reverb is warm, clear, and very programmable.

In addition to handling all of our the MIDI composition chores, the recently upgraded StudioVision Pro 3.5 has some essential digital audio features. Like Pro Tools, the recent update "went native" and is therefore much faster. Still, the most unique feature of StudioVision Pro is the ability



Digidesign's Pro Tools III 4.0.

Continued on page 27

ability to come up with creative solutions to these problems.

2. Every drop of energy that goes into being discouraged by the limitations of a particular project is energy taken away from making a great sound design.
3. Know your role on the team. Projects need to be driven by a singular, cohesive vision usually espoused by a producer, lead designer, or director. Unless you're working on an "audio only" product, audio is a supporting member of the cast; it doesn't lead the design. Audio is no less important to the overall success of the project; but, it follows and supports the design ideas and constraints defined by the project's singular vision. The sound designer should become comfortable in this role so as to avoid great heartache and suffering. However, this doesn't mean that there is no opportunity for creativity. (See Rule 4.)
4. This is the "two things" rule. Most of the time, you'll be taking direction from someone who knows less about audio than you do. By saying this, I don't mean to denigrate the skill of the project director; I'm just stating a simple fact. The sound designer is the expert when it comes to the details of audio. Yet the direction for the sound design must come from the person who is responsible for the project's overall vision. Otherwise the sound will not hang together with the product. My highly unscientific experience has shown that a project director is unlikely to have more than two identifiable design needs for any given part of the sound design. If you, as the audio designer, satisfy these two things, you're usually free to complete the bulk of the task with your full creative input. It's best to know what these two things are before any significant amount of work is done.
5. Run-time resources will always be shared among different disciplines.
6. As soon as the artists or programmers figure out how to use something effectively, it will no longer be available for audio (for example, the

Tools of the Trade (cont.)

to convert digital audio to MIDI and then back again. The best way to describe this is to give an example of how we use it. As you know, there is nothing like a live guitar track to "humanize" a MIDI composition. Unfortunately, it's seldom possible for us to record such a track and have any way to play it back as a digital audio stream. MIDI guitar tracks, which are recorded with a keyboard and played with a guitar sample, are more realistic, but tend to lack all of the subtle pitch gestures and tonal variations that make the track sound "real." Our solution is to record a live guitar track into StudioVision Pro. We convert that track to MIDI, which yields a stream of notes and pitch bend data that matches the performance. Next, we move all of the pitch bend data to another track and reconvert the MIDI data back into the digital audio track. This gives us a less interesting, but more easily looped set of guitar samples.



Yamaha VL-70m synthesizer

adds some much desired features. For one thing, it now allows you to use all of Wave's high-end plug-ins. You can also save job lists in order to keep track of and reload your files and settings from previous jobs. The job list is exported as a text file, which is great because you can get in and edit it by hand if you want to make a small change to a big job. If all of that isn't enough, they even include their own Waves IMA ADPCM (Adaptive Pulse Code Modulation) compressor, which simply kicks ass.

Synthesizers & Digital Audio Processing Software

I rely on four main units: Digidesign's SampleCell II, Kurzweil's K2500, my Oberheim Matrix 1000s, and the new Yamaha VL-70m. SampleCell is great because, as a NuBus board, it is integrated into the Mac's file system. There is no need to transfer files over SCSI (or God forbid) MIDI. I like to make my own samples, but I recently got a some great disks from ILIO's Heart of Africa and Supreme Beats series, which I'm using extensively on ENTER THE GECKO.

Continued on page 28

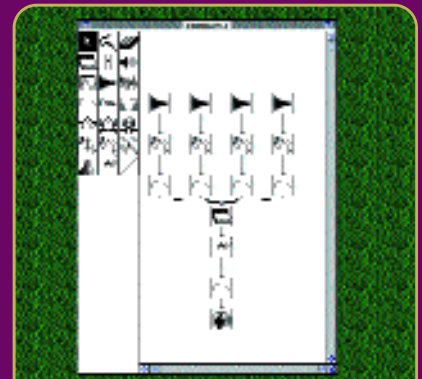


K.S. Wave's WaveConvert Pro.

Choosing the individual notes (samples) that have the most tumbrel variation, we build a compact, but varied guitar instrument. Finally, we play this instrument with the full MIDI plus pitch bend track and use program changes or velocity to switch between the different guitar samples, thus recreating (approximately) the original performance.

Audio Batch Conversion Tools

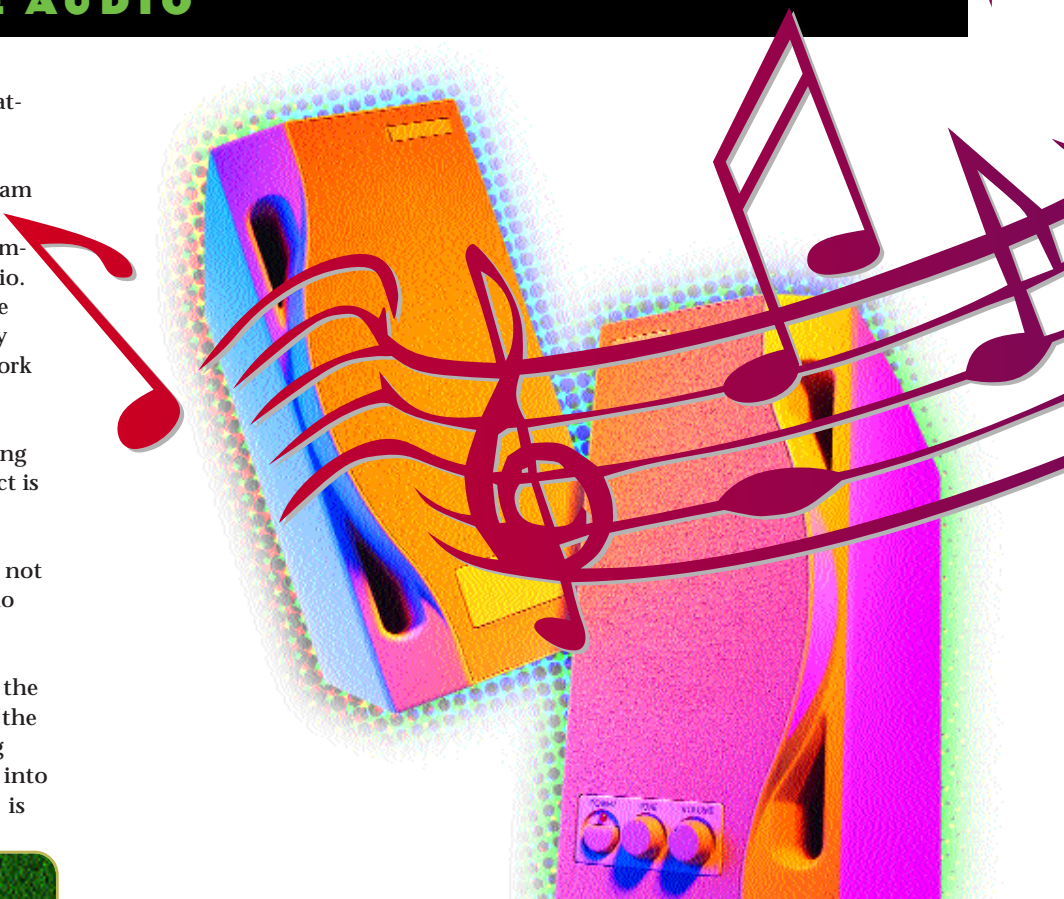
Wave's WaveConvert Pro is the latest update to WaveConvert, a batch converter that I often refer to as "the third member of my staff." The new version of the product is as easy to use and as solid as the old version, but it



Digidesign's Turbosynth.

CD-ROM drive on any game platform).

7. Making audio interactive is a team effort. The application must be altered by designers and programmers to support interactive audio. Team buy-in is essential because interactive audio, although very valuable to a project, is more work for everybody.
8. The likelihood of audio becoming interactive for any given product is inversely proportional to the amount of programming that's required of individuals who are not specifically assigned to the audio team.
9. It's far better to determine how the sound design will interact with the world before you begin creating assets. Retrofitting interactivity into audio designs, especially music, is



Antares Infinity digital audio processing software.

Tools of the Trade (cont.)

Since it's not easy for me in my limited space to record live players, I've been looking for a physical modeling synthesizer to add some human feel to my Red Book tracks. I found one

that I like in Yamaha's VL-70m. Physical modeling is a technique for creating sounds that is very different from sampling. In a physical model, rather than recording a single performance and looping it, you model the physical characteristics (length and diameter of a pipe; length and thickness of a string; size, shape and reflectivity of a body of a real or imagined instrument) in DSP. You can then tweak these parameters to get the nuances of the of the real thing. Although I've just begun to experiment with it, I've already made some outrageous saxophone patches.

On the software side, I spend a lot of time in Digidesign's Turbosynth and Antares Infinity. Turbosynth, though aging without much support, is a great sound effects program. It works like a modular digital synthesizer, allowing you to set up custom patches in the old patch cord paradigm. What I like about it is that you have flexibility in how you route things and that you can have very precise control over filter, amplitude, and pitch envelopes. You can even extract the amplitude envelope off of a sample and then use it as you wish.

Digital Looping Tools

Infinity is the definitive set of digital audio looping tools. The term "looping a sample" refers to defining a section of the sound data to repeat as many times as necessary to last a specified duration, as opposed to writing all of those repeats out longhand. When you have 512K of sound RAM to work with, compressing your instrument and sound effects data with looping technology may be your best and only option. Looping is a very tricky business, however, and there's nothing even close to Infinity's set of advanced tools.

Infinity's Rotated Sums Looper is especially well-suited to the PlayStation. The PlayStation has built-in ADPCM compression with a 28-sample block size. While this is essential, as it gives you, effectively, more space to store sound in, it also puts a limitation on your ability to loop sounds. Loop points must occur on sample numbers that are divisible by 28. Anyone who has looped small sounds before knows that it is hard enough just to get something that sounds decent, let alone to do so with such a draconian restriction. Fortunately, Infinity's Rotated Sums Looper allows you to get a rough loop with the loop points on the mathematically correct samples and then process the data in between the loop points until it is smooth and even without screwing up the math. How exactly this is done is more than I have room for, but trust me, it works wonders, especially for ambiance and other long, looping sound effects.

difficult at best, and severely compromised, if not impossible, at worst.

10. Leverage off of existing technology wherever possible. If you plan to create new audio technology, use off-the-shelf tools whenever you can. For example, I can't conceive of a scenario where it would make sense to write your own MIDI sequencer. Programs such as Opcode's Vision and Logic Audio are great tools. I can't even begin to speculate on how many person-hours went into making them. It would be crazy to invest development dollars in a "roll your own" sequencer. Rather, we need to create additional tools that map out the territory that is unique to our endeavor. Such tools should begin with the output of commercial tools such as a MIDI sequencer and add functionality as needed.

Creating an Adaptive Audio Engine

Armed with the desire for truly interactive audio, we at Crystal Dynamics set out to create our own sound driver for the Sony PlayStation and, perhaps later, for Windows 95. From my notions of interactivity and set of rules for interactive audio, we derived a number of design goals for our driver.

EMPHASIZE MIDI OVER DIGITAL AUDIO. For most of our products, MIDI and custom DLS-like instruments are a better way to go than Red Book or streaming digital audio. Rules 1 and 5 have some implications for Red Book audio. Red Book audio sounds great, but fortunately for Crystal Dynamics, our programmers know how to get the most out of the CD-ROM drive for game play. Therefore, it's not always available for audio. Furthermore, creating interactive sound designs using streamed, branching digital audio segments is limited in many ways, primarily by disk space, seek times, bus bandwidth, and buffer sizes. Red Book, or any kind of linear, streamed digital audio requires a lot of storage space in any situation. But it becomes even more problematic in an adaptive setting where each variation of a given section of music has to be remixed and stored uncompressed. Finally, most

Getting More Info about the IASIG

Interested in interactive audio? Point your browser to www.iasig.org. This is the web site of the Interactive Audio Special Interest Group. Look under "Working Groups" and the "ICWG" (Interactive Composition Working Group). Here, you will find the latest work of a dedicated group of industry professionals from companies such as Microsoft, Apple, Electronic Arts, Headspace, and Crystal Dynamics that are working toward unified, open standards for implementing interactive audio.

consoles (including the PlayStation) save money by using inexpensive (read, slow and not very reliable) CD-ROM drives. Thus, the constant seeking and playing of digital audio tracks is likely to tax the CD-ROM drive to the point of failure. Red Book audio should therefore be reserved for noninteractive sections of the game, such as title screens.

On the other hand, MIDI is small, compact, and easily modifiable on the fly. The PlayStation has its own dedicated sound RAM, and all of the data needed for a level can be loaded in less than a second. Once loaded, the data is out of the way and the CD-ROM returns to its other duties. Furthermore, the PlayStation contains a pretty good sampler. Respectable music and sound effects were created for the Super Nintendo as well, but that platform suffered from limited sound RAM. Fortunately, the PlayStation has almost ten times as much sound RAM, much better sound interpolation (an on-the-fly sample rate conversion technique used to stretch a sample up or down the keyboard from its native pitch), and superior DSP (used for reverb and the like). In my opinion as a confirmed curmudgeon, anyone who says that they can't make high-quality MIDI music on the PlayStation under these conditions is just whining about the amount of work involved in such an endeavor.

KEEP SOUND DRIVERS EFFICIENT. When we considered replacing the existing sound driver with our own technology, we decided that our code would need to be faster, smaller, easier to implement, and more capable than the code we would be replacing. Otherwise, the project was not worth undertaking. Rule 1 dictates that sound drivers must be small and fast, since both system RAM (where the driver resides) and CPU time are scarce commodities in a

fully rendered 3D world. Making sound drivers easy to use is important; programmer and game designer time are limited commodities, so making basic implementation easier leaves more time for these folks to work on making the world ready for your interactive audio.

There should also be a simple, consistent means for your game to communicate relevant information about itself to the sound driver. Adding interactive sound capabilities requires programmers and designers to spend more time communicating information to the sound driver about the state of the world and the characters within it. At Crystal Dynamics, we tried to remedy this situation by communicating the state of the world to the sound driver in the form of a set of simple, numerically registered variables. Most often, we use values from 0 to 127 so that they could be set from standard 7-bit MIDI controllers. Thus, the number of enemies alive on the screen might be represented as one 7-bit variable. Your distance from the level's exit might be stored in another. We have tried to use these same variables throughout the game so that they only need to be coded once.

CODE OR DIE. It's important to put the logic programming in the hands of the sound designer, not the game programmer. Rule 8 clearly shows the logic behind this. It's hard enough to explain which aspects of the world you need to track. It's almost impossible (and I think unreasonable) to expect the game programmers to write code to mute and unmute specific MIDI channels when various conditions arise. To solve this problem, we created (with some help from Jim Wright at IBM's Watson research lab) a programming language that allows us to author logical commands within a stock MIDI sequencing environment and store



them within a standard MIDI file. The language contains a set of fairly simple Boolean functions (**if then, else, endif**), navigational commands (**goto, label, loop, loop end**), a set of data manipulation commands (**get, set, sweep**), and parameter controls (**channel volume, pan, transpose**, and so on). Next, we created an auditioning tool that allowed us to simulate the run-time game environment, kick out logic-enhanced sequences, manipulate the game state variables, send commands, and see what happens.

Case Study: The GEX Project

30

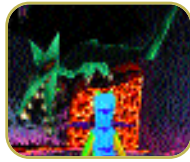
Most of my work this year has centered around the interactive sound design for our upcoming PC and PlayStation title, GEX: ENTER THE GECKO. (The PlayStation version will be the first title to use our new sound driver.) This game is the second installment of the GEX franchise and has undergone a complete technological overhaul by some of the best programmers and designers in the business. The game will include at least eight worlds thematically based upon TV and movie parodies, as well as secret, bonus, and boss levels — fertile ground for a sound designer indeed.

I'll talk briefly about what we're doing currently in the game's audio. For clarity's sake, I've focused on the treatment of a variable called **numberofenemies** and a small number of related variables. These few examples are by no means all that we plan to do in the game, but they illustrate my main points.

One aspect of the new GEX title is the game's fast and efficient engine. It allows us to have more enemies on screen moving at a faster rate. Since this is one of the features that really makes the product stand out, we on the audio team designed the interactive audio to work tightly with the new engine functionality. We set up a 7-bit variable called **numberofenemies** that reflects the number of enemies on the screen and is updated by the game continually. This variable is read and used by the sound driver to adjust the game audio.

Here's a breakdown of the GEX audio, by game level:

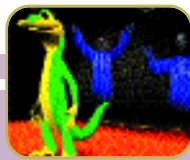
THE PRIMAL ZONE. The control track of



the MIDI sequence has logic built into it. When the **numberofenemies** register is **0** and **kills** is **1**, the sequence is paused, and playback of all tracks is branched to a short stinger of the GEX theme song orchestrated with the instrument palette of the Primal Zone level. When the stinger is over, the main sequence is resumed.

We keep a **location** register that allows the sound driver to see if GEX has entered a new area of the map that has different music. The control track checks every four beats to see if the value of **location** has changed. If it has, a short transition drum fill plays, and then the MIDI sequence that is matched with value of **location** is started on the beat.

SCREAM TV. Scream TV is the "horror"



level of ENTER THE GECKO. For this level, I used eerie chamber music laid over some ambient loops of slow ragged breathing and strange heartbeats. The same **numberofenemies** register is used in this level, but instead of muting and unmuting, the value of the register affects the harmonic center of the piece. For this type of transformation, we use a technique called "pitch mapping." A pitch map in our system is a mechanism to remap the pitches of individual notes as they come from the MIDI sequence on their way to the physical voice. Our pitch maps are built so that the transformation can be kept within the prevailing harmony of the piece. In other words, for a scary piece such as this, the pitch table will confine all of the note remappings to a diminished scale. The **numberofenemies** value is used continuously to set the number of scale degrees upwards to transpose the harmonic instruments. This creates a very smooth, subtle, intensifying effect as more and more enemies are on the screen and danger is increasing.

CIRCUIT CENTRAL. The Circuit Central



level looks like the inside of a CPU. The opening music is weird, ambient analog electronica with a slow, trip-hop

beat. After some exploring, it becomes clear to the player that there are jumps and expanses that cannot be passed in Gex's normal state. Within this time, the player should also discover a number of chargers set out in various locations. When Gex steps into a charger, he starts to glow in a green light with orbiting electrons. This state lasts for 15 seconds and gives Gex the ability to use embedded "chips" in the floor to do super jumps and turn on "data bridges" that span the formerly uncrossable chasms.

The state of being charged up is stored as a **1** in a **chargedup** register, and **location** keeps track of the seven different areas within the main level. When the control track in the main sequence sees a **1** in **chargedup**, it checks **location** and matches the area to one of seven different 15-second-long high-energy pieces of audio.

While most often the game affects the sound design, this relationship can also work the other way. In Circuit Central, we set a register at every beat of the music. Since each measure has four beats, I set the **beats** variable with a **1** on the down beat followed by **2, 3**, and **4** on the remaining three beats and then cycle back to **1**. This timing information will synchronize the lighting effects in this world.

FOR FURTHER INFO

Antares

<http://www.antares-systems.com>

Digidesign

<http://www.digidesign.com>

Ilio

<http://www.ilio.com>

K.S. Waves

<http://www.waves.com>

Kurzweil

<http://www.youngchang.com/kurzweil>

Oberheim

<http://www.gibson.com/products/oberheim>

Opcodes

<http://www.opcode.com>

Yamaha

<http://www.yamaha.com>



Only the Beginning

Adaptive audio is a very new field. Many challenges lie ahead, but I firmly believe that it represents the future of sound in interactive media. To move forward, we need a major paradigm shift in how we think about music, great tools, new technology, and a healthy dose of realism. I hope that my anecdotes, rants, and factoids have shed some light on and sparked more interest in creating interactive audio. So long for now from audio central at Crystal D. ■

Mark Steven Miller has been producing audio for interactive media since 1989, when he started Neuromantic Productions. Neuromantic Productions produced audio for over 60 titles for publishers such as Sega, Acclaim, Electronic Arts, and Virgin Interactive Entertainment. Since closing Neuromantic in 1994, Mark has served as Audio Director for Sega of America and served as the Working Group Chairman of the Interactive Composition Work Group of the Interactive Audio Special Interest Group of the MMA (IA-SIG). Currently, Mark is the audio and video director for Crystal Dynamics and serves as the Co-Chairman of the IA-SIG. Upon completion of his work on GEX: ENTER THE GECKO, Mark will be taking the position of Senior Producer at Harmonix Music Systems in Cambridge Mass.

32



CREATING AN INTERACTIVE AUDIO ENVIRONMENT

34



Audio in today's interactive entertainment media has progressed far beyond the bleeps of early video games. An object or an environment within a game exhibits a number of complex relationships. A creature may be surprised to see you. A robot's gears get stuck when it tries to move toward you. A diabolical enemy is afraid of the dark. When encountering these elements in a game environment, we expect them to communicate to us through audio in subtle and different ways. Aspects of emotion such as surprise, frustration, admiration, and fear could easily be conveyed through an enhanced and well thought-out object vocabulary.

BY DANIEL BERNSTEIN

Our lives are full of an ever-present collage of audio cues that we take for granted. For example, at this “quiet moment,” I can hear the cascading sound of a fountain in a pond, the intermittent quacking of ducks and geese, a baby in the background, someone pouring a bucket of water outside, and a plane flying overhead. All of these cues, though subtle and seemingly unimportant, create the ambience of a particular scene, imbuing

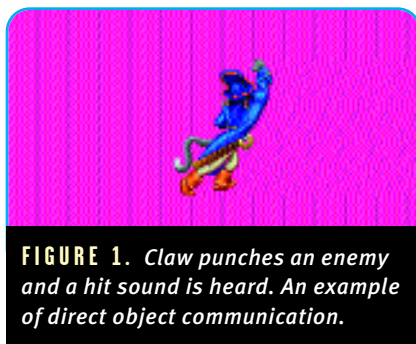


FIGURE 1. Claw punches an enemy and a hit sound is heard. An example of direct object communication.

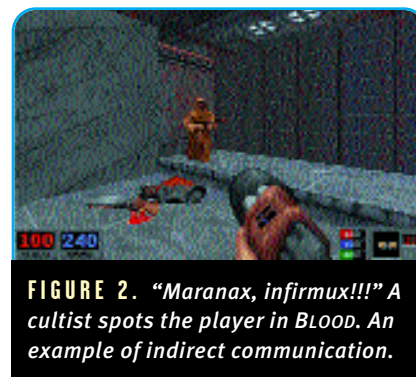


FIGURE 2. “Maranax, infirmux!!!” A cultist spots the player in BLOOD. An example of indirect communication.

AN INTERACTIVE AUDIO VOCABULARY FEATURING A WIDE ASSORTMENT OF SOUND CUES AND BACKGROUND NOISES IMMERSSES PLAYERS FAR MORE THAN YOU MAY REALIZE.

ing it with identity and significance. Without these background sounds, or ambiences, our lives would sonically resemble a lunar landscape. A collection of sound cues such as this within a game environment refers to the non-causal relationship of a player to the game. The sound space isn't triggered by the player's direct action. Instead, the sound is affected by and reacts to the environmental aspects of the scene that is being conveyed.

When we go to a movie, our emotional response is directly related to the music. The music swells, our anticipation grows, and our adrenaline rushes. The music ebbs, and we feel a calming sensation. This is very easy to convey in a linear medium, where the ending and the progression of events in a movie is predetermined; but how do we compose a soundtrack to a game if it can follow many paths and endings? An adaptive soundtrack that responds well to game events is one of the best ways to envelop the player in a game experience.

Audio Object Vocabulary

An audio object vocabulary is a method by which game objects (not necessarily just speaking ones) talk to each other and the player. The methods of communication vary from

object to object and from context to context. There are three types of object interaction: direct, indirect, and environmental.

DIRECT COMMUNICATION. An object communicates directly as a cause of direct action on its part (Figure 1). When the ball hits the paddle in the old arcade game PONG, it makes a bleep. This is direct object interaction. Unfortunately, most games haven't explored far beyond this simplistic level of object interaction. Direct communication is important when you want to convey specific audio cues, such as a scream of pain when you shoot a monster, or the creak of a wooden rocker when you push back a rocking chair. In Monolith Production's CLAW, I found it important that every character had something different to say when you interact with him or her (or it), even if it's in combat. For example, a melodramatic character, while dying, would say “I'm dying... I'm dying... I'm dying... I'm dead,” with an animation to suit. A more primitive character would emit a squawk, and a more substantial enemy would yell out, “I curse you, Claw” as he falls to his death. When you hit a lounge-lizard-turned palace-guard-merman, he would say, deadpan, “Ouch that hurt quite a bit.”

As always, a variety of audio cues are paramount in ensuring that a set of

quotes doesn't become repetitive. From a programming standpoint, that may require a bit more intelligence to pick out the quotes. A buffer with an index to the most recently used quotes helps a lot because it shields the player from experiencing the same “random” set of sounds in rapid succession.

INDIRECT COMMUNICATION. This is an indirect method of object interaction. That is, by causing something to happen in the game, something else responds sonically. A typical example of this is a “sighting” state for an enemy. When an enemy sees you, and his or her AI changes, a sonic cue that signifies that change may be appropriate. In Monolith's BLOOD, for example, cultists scream in a terrifying foreign language (created for the game drama) a series of epithets when they spot the player (Figure 2). In CLAW, every enemy has something different to say in the “sighting” state. A female boss taunts Claw in a mildly suggestive manner when they come into contact. A goofy bear sailor exclaims “I don't like you” when he sees a player.

Other sonic cues may convey indirect object interaction. Your character may begin breathing heavily when he or she is tired (health is less than some coefficient). Your metal body suit emits a rubbing, squeaky noise that signifies rusting. In addition to sonic cues that help convey complex visual phe-





FIGURE 3. "Umm, there's nothing here to eat and I'm kinda hungry." An example of environmental communication.

nomena, certain characters within the game display behaviors that can be conveyed easily through sonic cues, even if they aren't represented visually. Indirect cues can be based on a number of different motivating factors, the rules of which can be determined at the game design stage. For example, in Blizzard's WARCRAFT II, clicking on an ax-throwing troll more than once causes it to respond with annoyance, even though no animation is being shown. This is highly effective character enhancement.

ENVIRONMENTAL COMMUNICATION. A character or object in the game may generate a system of audio cues on its own, irrespective of its communication to the player. This is purely a function of a character's existence in its environment. It may be busy chatting to itself or other characters. It may generate a sound or a series of sounds on its own. Our goofy bear sailor from CLAW will comment on how hungry he is or where his pet rat might be when he's in an idle state (Figure 3). Depending on where he is in the game, Caleb (the character you play in BLOOD) may pick from a variety of different show tunes to sing while he's taking a break from the carnage. A thespian tiger from CLAW recites different Shakespearean passages as he muses on his own omnipotence.

Environmental communication need not be comic, nor does it need to be vocal. A swishing blade and a humming motor sound signifies an industrial fan in BLOOD, while a phone may be ringing intermittently. A character may pass by an alien hive, with pods emitting a terrifying whine.

Environmental communication is paramount in reinforcing a character or object's existence in the game environment. The character literally comes alive as a personality or physi-

cal entity. But as with all different types of object interaction, it's important to remember to keep a consistent set of sounds from character to character. In CLAW, I made a decision to use three different idle cues (environmental communication), four different sighting cues (indirect communication), and between eight and nine sounds (direct communication) to describe each character sonically. In the end, most characters used more and some less than that average. However, planning the audio object vocabulary ahead of time helped to maximize the use of memory allotted to sound in the game.

Character Development

The nature of a game object must be relayed in the character of its "voice." It's very easy to screw up the integrity of a character by giving different visual and aural personalities. However, giving the right "voice" can greatly enhance a character's personality. A weak character may be depicted through the use of a humorous voice. A stronger character's dramatic personae can be highlighted through the use of a deeper and more resonant voice, as well as a script that relates without question his or her authority.

TIPS AND TECHNIQUES.

1. Always use professional voice actors. Trained voice actors are professionals who specialize in giving your character the voice it deserves. Whether it is a cartoony or a deep resonant voice, a single talented actor may help develop your ideas for multiple characters and realize them in ways that you haven't conceived. When in doubt of how to find a good voice actor, look to talent agencies and talent search services for help. Moreover, making a trained voice work with the rest of your mix is quite a bit easier than trying to amplify or equalize a weak voice. All sound engineers can attest to this.
2. Spend a little more time in sound design. As in all cases, don't just pull sound effects off of a CD. Create sound effects from your own sampled sounds as much as possible. A portable DAT recorder and a good microphone in the field will

take you much further than a commercial CD sound library ever could. Nothing kills a unique audio environment more quickly than the phrase, "I've heard that somewhere else before...."

3. Collaborate with professional scriptwriters. Writers would jump at the opportunity to write a couple of hundred lines of dialogue for some game characters. The results will definitely be worth the investment.
4. Don't be afraid to inflate the vocabulary. Minimize silent time. If you have the space for audio, use it. Set the limit with the programmers and designers early as to your memory budget for audio, and use it wisely.

Ambient Sound

Ambient sound refers to the sound world that is generated from a player's location in the game space. It is a system of indirect and environmental cues that immerse a player in a particular setting. As in my real-world example, we are surrounded by ambience all of our lives — a complex web of sound. However, ambience is the most underdeveloped side of sound design in interactive media. A game with little or no ambient sound presents little or no connection to how we perceive the outside world with our ears. An ambient sound world might be as simple as a single looping track of forest sounds or a system of sound-producing objects all linked together by their location within a given game environment.

The environment can communicate to the player information important for the game-playing experience. For example, a raven flies by in a forest, making a screeching sound that informs the player that he or she has ventured too far. A swamp makes a menacing gurgling sound, informing the player that he or she shouldn't go there. The sound of a portal opening and closing in the distance informs the player that he or she is close to the level's exit.

Environmental ambiances fully transport a player into the world presented by the game. In CLAW, each level has a distinct set of ambient sounds based on the terrain that the main character is encountering.

Within a terrain, a single (environmental) looping sound is used (such as the sound of a forest), along with a set of sounds (indirect cues) that are triggered either by Claw's location on the map or by random chance. For example, the sound of a character whistling in a window matches the animation of the character shaving and the background ambience of village noise. When Claw moves through another terrain, the looping ambient sounds would cross-fade, and another set of ambient trigger sounds would be selected that corresponds to the new terrain.

In *BLOOD*, I used ambience to enhance the atmosphere, as well as to connote physical environments. In a

use trigger ambiances whenever possible. Trigger ambiances help mask the loop point, as well as provide overall variety in the ambience. In Kesmai's *AIR WARRIOR*, I used trigger ambiances to convey the sound world of a World War II airfield. During any given time, an airplane fly-by sound, a vehicle drive-by sound, and an airplane startup sound would be selected and played from a set of 50 or so trigger ambiances. Since these trigger ambiances were selected randomly and played at random times, the sound world was always changing and seldom repetitive. Another method of avoiding loops is to queue similar sounds one after another. A



FIGURE 4. A dark temple emits a menacing chant.

The inclusion of atmospheric elements adds to the **spooky and scary nature of the game's look and feel.**

temple, distant chanting is heard (though the source of the chant is never discovered) (Figure 4). In a narrow hallway, whispers surround the player from all sides. The inclusion of atmospheric elements adds to the spooky and scary nature of the game's look and feel.

TIPS AND TECHNIQUES.

1. Try to use consistent reverb settings. All sounds within a given environment should have a similar set of reverb settings that place the entire sound world within a consistent acoustic space. There are foreground and background elements that do stand out from within the ambience, but not so far as to mistake these sound elements for characters or objects that a player must encounter.
2. Make your loops seamless. The looping ambiances in the game need to be smooth and unnoticeable. Large variations in pitch or amplitude will make the loop quite recognizable and annoying after a while. A rhythmic pattern works well (like the sound of crickets), if it's cut perfectly. Also, a longer sound sample will help mask the loop point.
3. Avoid loops. Though seamless loops are not an impossibility, it's best to

set of three or four sounds that fit seamlessly end-to-end will work well if they are selected to play on a single channel randomly. This helps break up the pattern created by a single looping sound.

4. Try to create fine gradations of ambiances. Say we're walking from a forest into a mountain pass. We start out in a deep forest then walk through a leafy forest then into a meadow before reaching the mountain pass. If we have a single sound for the forest ambience, no matter how the forest changes, the ambience will remain the same until we change scenery drastically when we reach the mountain pass. However, if we subdivide the forest into three gradations (deep, leafy, meadow), we'd be better able to convey to the listener the transition of environments from forest to mountain pass.

Adaptive Music

The nonlinear medium of computer gaming can lead a player down an enormous number of pathways to an enormous number of resolutions. From the standpoint of music composition, this means that a single piece

may resolve in one of an enormous number of ways. Event-driven music engines (or adaptive audio engines) allow music to change along with game state changes. Event-driven music isn't composed for linear playback; instead, it's written in such a way as to allow a certain music sequence (ranging in size from one note to several minutes of music) to transition into one or more other music sequences at any point in time. An event-driven music engine must contain two essential components:

- Control logic — a collection of commands and scripts that control the flow of music depending on the game state.
- Segments — audio segments that can be arranged horizontally or vertically according to the control logic.

In Kesmai's *MULTIPLAYER BATTLETECH*, control logic determined the selection of segments within a game state and the selection of sets of segments at game state changes. Thus, the control logic was able to construct melodies and bass lines out of one to two measure segments following a musical pattern. At game state changes, a transition segment was played, and a whole different set of segments was selected. However, this transition segment was played only after the current set of segments finished playing so as not to interrupt the flow of the music. I selected game states and also tracked game state changes based on the player's relative health vs. the health of the opponent. Overall, I composed 220 one to two measure segments that could all be arranged algorithmically by the control logic. What resulted was a soundtrack that was closely coupled with the game-playing experience.



TIPS AND TECHNIQUES.

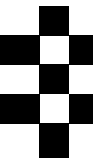
1. Music comes first. Remember that no matter how closely your music follows the game play and how interactive it is, if it doesn't gel as a musical composition, you're better off writing a linear score. Always explore all possibilities of transitions from one game state to the next, and see if the music reacts the way you meant it to react. Make sure that you write transition sequences and that the engine is intelligent enough not to change game states midmeasure or midphrase.
2. Decouple segments horizontally and vertically. Compose your music so that different segments may be combined end-to-end (horizontally), as well as on top of each other (vertically). This way, you can combine different melody lines with bass lines, use different ornamentation, and so on.
3. Don't give away too much information. Sometimes a musical cue might say too much, when it was meant just to highlight the game state change. For example, in a certain game, an upward chord progression always signifies to a player that a starship is on his tail. When working on game state changes, make sure your event-driven music isn't used as an early warning system for the game.
4. Define a series of translation tables to track game state changes. For example, in MULTIPLAYER BATTLETECH, a game state change from "winning" to "advantage" implies a losing trend. The music reacts to this change by selecting a different set of segments than it would if the change occurred from "advantage" to "winning."

By composing in a nonlinear fashion, and by having the music react to the player's actions directly and indirectly, we introduce a new level of interactivity. Emotionally, the soundtrack carries the person seamlessly along with the action in much the same way as the static, linear media of film. In this fashion, music becomes the gateway to the player's emotional response to the game.

Total Immersion through Sound

As game designers and audio producers, we should be constantly aware of the impact that a well thought-out audio environment can have on the product. It can make a graphically simple and uneventful scene become awe-inspiring. Effective use of an audio object vocabulary can enhance the impact a character may have on the game player. Ambient sounds, in all of their variety, can transform a game scene from a virtual one to a believable one. Surreal textures and atmospheric gestures can generate emotional responses in a player as varied as the soundscapes themselves. As games become more and more complex and graphically spectacular, we must not overlook the role of audio in enhancing and completing that feeling of total immersion. ■

Daniel Bernstein manages Monolith Productions' Audio/ Video department in Kirkland, Washington. He has been the audio producer, sound designer, and composer on a number of successful commercial titles, most recently for BLOOD. When not listening to feedback loops, he enjoys spending time with his wife basking in the beautiful Seattle sunshine. He can be reached at dbernstein@lith.com. Please, no e-mails after 10PM.



From LEDWARS to LMK: Learning Structuring Data

40



that I coded each rolling barrel coming down the ramps separately for every level. You can imagine the fun I had checking to see if the player collided with one of these barrels. This was my first lesson in data structure design, and it was a valuable one.

How you design your data structures can affect your game significantly. With careful planning, your data structures can provide substantial performance gains. Throughout this article, I will use two games that I've been working on this year as examples of thoughtful data structure design. The first is LEDWARS, a real-time strategy game (Figure 1); the second is THE LADY, THE MAGE, AND THE KNIGHT (LMK), a role-playing game (Figure 2). LMK is a multiplayer game set in a huge game world that you can roam around freely. There are no levels, as the entire map is connected. This brought up some interesting data structuring problems, which I'll explain shortly. Although both LMK and

LEDWARS are completely different types of games, they share a good deal of code. This reusability was possible in part because I approached the development of LMK (which I started first) in an object-oriented way.

Design Goals for LEDWARS

If there's one thing that's important in a strategy game such as LEDWARS, it's the intelligence of the computer opponent. As artificial intelligence usually boils down to looking at loads of data, this means that the data has to be easy to manipulate, and in a suitable format for the AI engine to use. The problem is that usually the best AI algorithms are created once the game is finished (and indeed they were in this case). Therefore, it's crucial that you can easily change AI routines at any point in development without impacting the rest of your game. Since LEDWARS is also a multiplayer-

When I was 12 years old, I wrote a DONKEY KONG clone on a Sinclair ZX81 that had a 16K RAM expansion cartridge. The game was embarrassingly slow, but not entirely because of the Sinclair's speed. Part of the problem lay in the fact

er game, the data structures needed to handle this potentially painful part of the design. I wanted to design the objects in the game so that they could be handled the same way, regardless of whether they were controlled by the player or the computer. I also wanted to make sure that the amount of data necessary for the AI to make decisions was minimal, so that communication channels were not deluged with data. Finally, LEDWARS needed to be able to deal with many game objects at the same time. When played in the "world war" mode on a large map, the game can contain thousands of player units and buildings. If all of this data was handled inefficiently, the game's performance would slow to a crawl.

MULTIPLAYER CAPABILITIES. A multiplayer game needs some kind of messaging system, so I had to design one. I felt that the game's code shouldn't make any distinction — beyond the user interface controls — between local players



FIGURE 1. LEDWARS, a real-time strategy game



FIGURE 2. THE LADY, THE MAGE, AND THE KNIGHT (LMK), a role-playing game

HOW YOU DESIGN YOUR DATA STRUCTURES IMPACTS

YOUR GAME'S PERFORMANCE AND ULTIMATELY THE

REUSABILITY OF YOUR CODE. by Swen Vincke

and remote players. This code should send messages to a central data stream, which receives messages coming from all other network clients and AI opponents. LEDWARS used a peer-to-peer system, which means that all of the code should be determinant. (Determinant means that given the same starting conditions, the game flow should be the same on every machine in the network.) For instance, an AI opponent on one computer cannot do something other than what an AI on another computer is doing.

ARTIFICIAL INTELLIGENCE. I won't talk much about the AI in LEDWARS, as it would take up an entire other article (you can read about the pathfinding system in my article "Real-Time Pathfinding For Multiple Objects," pp. 36-44, *Game Developer*, June 1997). Instead, I'll touch on how I managed the AI given the design goals.

An adaptable AI engine requires that you organize data in a straightforward way. If a lot of filters are necessary to generate additional information, you might have to change the filters every time something changes in the AI code. Because so many units can be in the game simultaneously, our AI needed to be able to remember previously calculated results. It couldn't afford to

reanalyze a situation every time it moved a unit or constructed a building. Since situational analysis (for example, assessing where the largest threat is) takes significant processor time, this section of the AI code had to be abstract enough to last multiple frames while a new situational analysis was under way. By "abstract enough," I mean that the AI really shouldn't bother with things such as "This enemy unit is here on this hill. If it moves a little bit to the right, it'll be able to shoot the hell out of our unprotected facilities." Rather, the AI's consideration should be, "Look, this area, which is of some importance to us, has some threat to it." An AI is better able to deal with nonspecific information; the AI doesn't have to track changes in a situation for every frame. In the example, it is enough that the AI knows that there is threat in this area. Even if the enemy unit moves a bit, the threat remains. Given all of these considerations, I split up the AI into three parts.

The master AI receives most of its data directly from game objects (for example, player units, buildings, and so on) and "sensors." The game objects contain data such as the resources that they require, their team morale, and so on. The sensors, on the other hand, are

like satellites. They fly over the map slowly, gathering strategic information such as the importance of certain geographical areas, the threat of enemy troop formations, and where the computer should build its next strategic facilities. The master AI processes the information that it has acquired from the sensors and game objects and applies this information to its objectives. It proceeds by giving commands accordingly. This approach met all of the goals I had set forth:

- The AI has a good overview of the situation at both the macro and micro levels, using its sensors and the information the game objects give it.
- Since game objects such as units and buildings supply data to the main AI, I have no need for additional structures other than those that already exist.
- Modifying the AI simply requires changing the rules that govern its decisions. I don't have to rewrite the AI engine.

HANDLING DATA. As far as handling a large number of game objects, this problem was solved by distributing the workload for handling all objects over several frames. Only the visual aspects (such as animations and movement) are comput-



ed for every object every frame. If something more processor-intensive needs to be done, the object has to wait until its turn. That way, a thousand units can be handled over a single frame or twenty frames, at a processing rate of about fifty objects per frame. In this manner, it's practically impossible for the player to notice any type of stall in the game during computations. Once the data handling section was completed, LEDWARS was actually a fairly straightforward game to develop.

Tiles as Data Structures

As a real-time strategy game, LEDWARS needed a map with objects moving on it. Since the maps that the game requires are too large to

use completely rendered backgrounds, tiles are used instead. As all game objects were going to be built from tiles, I decided to store collision information within the data structure of tiles. Listing 1 shows the structures used for this, as well as some of the other structures discussed in the next paragraph. You'll note that structures don't contain pointers to the structures they are made up of, rather they contain indexes to a global array that contains all these structures. This is convenient because if you store your structures to disk, you don't have to start figuring out where all those pointers are pointing.

LEDWARS also needed animations and tile groups. An "animation" is just a sequence of tiles, each of which have different appearances and contain different collision information. If a tile is

animated, a flag is set in the information field of the tile that also contains the collision information. A global array, which is the same size as the number of tiles, directly remaps tiles to a global array of animations. A "tile group" is a collection of tiles with an *x* and *y* dimension. The next level in the object hierarchy is the "tile group collection," which is a series of tile groups. These different structures were necessary to manage individual units, which consist of a tile group collection and some unit-specific data. Likewise, buildings consist of a tile group collection and some building-specific data.

Since a lot of units and buildings in the game use the same data, I split up their information into "static" and "dynamic" data. Static data never changes for a particular unit type, and dynamic data changes and therefore must be allocated for every unit. By categorizing data in this manner, I could create unit parents containing the static data for every unit and some default dynamic data. A new unit in the game copies the data from its parent, and then fills in the dynamic data as needed. The same goes for the buildings. Finally, a "map" is a collection of units, buildings, and tiles (see Figure 3).

LISTING 1. *The structures of game objects.*

```

Tile definition:
typedef struct {
    long ImageIndex ;//Index to a list of images
    long Info ;//Bitpacked information like for instance collision information
                //Also contains a bit which indicates if this tile is animated.
                //There exists a global array which remaps tiles to an animation(See
                next definition)
} TTile ;

Animation definition:
typedef struct {
    long AnimationSize ;//Size of the animation
    long *AnimationList ;//Contains indices to a global array of Tiles
} TAnimation ;

TileGroup definition:
typedef struct {
    long XSize,YSize ;//Dimension of the group
    long *TileList ;// Contains indices to a global array of Tiles
} TTileGroup ;

TileCollection definition:
typedef struct {
    long AmountOfGroups ;//How many groups are there in this collection
    long *GroupList ;//Contains indices to a global array of tilegroups
} TTileCollection ;

UnitParent definition:
typedef struct {
    //Static data
    long TileCollection ;//Index to a global array of TileCollections
    Game specific unit data
    Dynamic data
    Default values
} TUnitParent ;
    
```

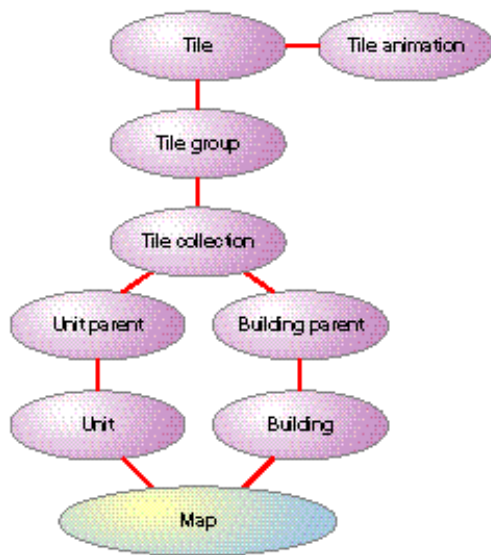
(Continued on page 44.)

Design Goals for LMK

While LEDWARS was a relatively straightforward game to develop, LMK was new territory. In LMK, the game world is highly detailed. The player has to be able to interact with every object — books, cupboards, magical potions, and so on — and every nonplayer character (NPC). Players have to be able to use objects, to drop them, to use objects on one another, and to change these objects. It was therefore important to design the data structure of game objects carefully. Here's how it was done:

- Objects are made up of a series of bit-packed information chunks that indicate what a player or NPC can do with that object. For example, if a player uses a torch, it starts burning, but to use a piece of meat, the player eats it.
- Objects can contain containers, which in turn may be made up of other containers. A cupboard, for

FIGURE 3. Object relations for LEDWARS.



instance, can contain a chest, which in turn can contain a bag, which can contain a pouch, which may contain one gold coin.

- Each player's (or NPC's) inventory consists of objects or containers, which affect the character's statistics. If there's part of a mutilated body in the bag you are carrying, then your charisma goes down.
- NPCs can interact with the objects and with each other. For instance, they can talk to each other or exchange objects.
- Players can question NPCs, and depending on the player's behavior during the conversation and also outside it (such as when you steal something from somebody), NPCs will react differently, thanks to highly detailed NPC scripts that can change dynamically.

All of these facets wouldn't have been too difficult to deal with from a development standpoint, but problems arose as a result of two design decisions made early on: to construct the game as one continuous level and to allow multiplayer functionality.

Developing the game quickly became complicated. For instance, if a player drops an object, every player in that game has to be immediately updated with that information, no matter how far from that action they are. The nature of the game play is such that players have to rapidly switch between the several characters belonging to their

parties. Thus, the world has to be synchronized. It isn't possible to delay these updates until a player switches to a character near the action, because the updates delay the character-switching process, and switching between characters is a common aspect of the game. To make matters worse, LMK's game world is very environmentally diverse (for example, the south pole and equator don't look alike), necessitating a great deal of different background art. Characters and monsters typically require 200-300 frames of animation, and

the average humanoid is 150 pixels tall. We needed a solution that would solve all of these problems.

Necessity, the Mother of Efficient Programming

These challenges necessitated an extremely creative design for my data structures. First, I needed a smart image manager that required minimal disk reads and yet kept within memory constraints — a system that would purge data from memory if it wasn't immediately needed. The game also required smart clipping features — there's no way the entire game world could exist in a computer's memory, yet I couldn't sacrifice speedy scrolling around the world. Faced with this constraint, I designed an image manager that determines the popularity of certain images. Popular images are the hardest to free from memory, while those that are seldom needed can be discarded very easily. Since block reads are a lot faster than separate seeks for every block that you need, the image manager also tries to anticipate what it

LISTING 1 (CONT.). The structures of game objects.

```

Unit definition:
typedef struct {
    TUnitParent *Parent ;
    //Dynamic data
    Game specific data
} TUnitType ;

BuildingParent definition:
typedef struct {
    //Static data
    long TileCollection ;//Index to a global array of TileCollections
    Game specific unit data
    Dynamic data
    Default values
} TBuildingParent ;

Building definition:
typedef struct {
    TBuildingParent *Parent ;
    //Dynamic data
    Game specific data
} TBuildingType ;

Map definition:
typedef struct {
    unsigned long XSize,YSize ;//Map dimensions
    unsigned long AmountOfUnits ;//Amount of units in map
    TUnitType *GameUnits ;
    unsigned long AmountOfBuildings ;//Amount of buildings in map
    TBuildingType *GameBuildings ;
} TMapType ;
  
```

will need and when. Based on this information, it tries to read as many images as possible from disk to memory in one long (but not too long) block read. The more memory in a player's machine, the more images the system caches. The same goes for the part of the map that is in memory. Depending on the speed and available memory, the system takes as much of the map as it can, leaving a lot of the map in compressed form until it's needed.

A great deal of data abstraction and data compression techniques were required to solve problems relating to distances between characters and far-away objects. LMK's map system can do operations on objects throughout the game world without having to know much about these objects. For instance, suppose the player thinks that it would be a good idea to block the entrance to a house with a table. An NPC wants to enter that house, but can only do so by moving the table. To enable this maneuver, the system first needs to know that there is a table blocking the NPC, that the NPC is strong enough to move the table (for which it also needs to know the table's weight), and where it can safely move the table. If the current window on the map is far away from the location of the offending table, that's a lot of things the game needs to know. To make matters worse, it's possible that there are items laying on the table that will fall if the NPC moves the table.

LMK addresses this problem by using several layers of information abstraction and working on these rather than directly on the world data. Once the part of the world where the table stands is read in, these layers are blended with the world again. One such data abstraction layer contains the total weight of a number of objects standing on a particular tile. Based on this compressed data, the system decides whether a NPC should move around an object or just move the object itself. Another layer, which divides the map in regions, remembers which objects should be moved. By doing some calculations using this latter layer, the system can effectively move the table with everything on it once that part of the world is loaded into memory (Figure 4).

The same goes for the characters; the characters have huge data structures because of all the actions that they can

take on objects within the game (and all the nice things they can do to one another). Rather than extrapolating the results of certain actions in real time, LMK calculates what a character is likely to do when it approaches an object (for instance, eating every single piece of meat in a city), saving a lot of computing power. Once again, using abstraction layers is the method. And again, just as in LEDWARS, distributing the workload over several frames helps a lot.

Since LMK is a multiplayer game, it needed a messaging system like the one in LEDWARS, with the difference that LMK required a client/server architecture — at least to a certain degree. LMK's objects are managed in a peer-to-peer fashion, so that if you move an object, all of the other clients update their view of the game world with that movement information. However, if you kill a monster, only the server is updated. In this sense, characters and objects are treated differently by the system. The server tells the clients what characters they see and what they will do. When an NPC nears the current window of a local player, the server sends a packet to the client telling it that this NPC is coming up along with information that describes its intentions.

For instance, the server can tell a client something like, "Listen up. This murderous NPC named Clovis is going to open the door you're staring at. If

the player is still looking at the same spot in n frames, you will show Clovis moving towards the door, opening it, and entering to kill everybody who is inside. So you'd better start loading in the animation frames for Clovis and get to it. If the situation changes, I'll tell you on a need-to-know basis. And for God's sake, don't send me a 'Didn't receive packet' message, because I'm too busy with a bunch of lunatic players who think slaughtering a town is fun, and now I'll also need to tell everybody that a door has opened." Using this sort of system actually lightens the network communication load!

Different Games, Similar Data Structures

At its core, LMK uses three types of images: tiles, objects, and characters. Object and character imagery have bounding-box collision information, while tiles have simple 2D bit-encoded collision information. Animations can contain any of these three image types. Tiles consist of image and collision information. Objects are collections of animations or images, plus specific object information. Characters are collections of animations and specific character information. A map is a collection of characters, objects, and tiles.

FIGURE 4. LMK deals with the huge world map using data abstraction layers. When parts of the map are read in, the layers are blended with the real game map.

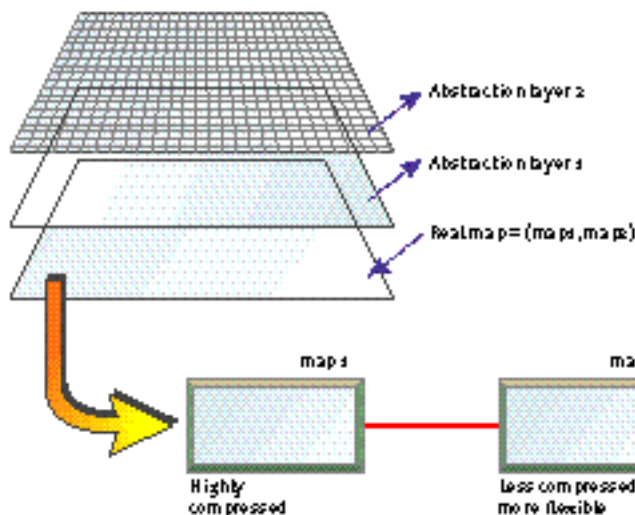


Figure 5 shows these relationships.

Sound familiar? It should — LMK's data structures are nearly identical to those found in LEDWARS, despite the fact that the two are different types of games. Because of their similarities at the data structure level, I could use virtually the same level editors for both games, and a significant portion of common code was shared between the two. One aspect of game development that most novices underestimate is the amount of work necessary to build the application frameworks for a game. You need tools to convert your artwork into a format suitable for your game, you need editors to create your game worlds, and you need a whole range of (usually) home-brewed utilities to facilitate the filling in of your data structures. The quality of a game often depends on the quality of the underlying application framework. For instance, if you are creating a real-time strategy game, and you find it difficult to change the position of a laser turret, you'll be discouraged to do so even if it will improve game play.

The problem with creating application frameworks, however, is that it's one of the more tedious parts of development. (At least I think so. But I'm sure a lot of database developers out there just love building them.) You can minimize the amount of time that you spend developing application frameworks by reusing them across titles, as I did. I developed LEDWARS while I was busy with LMK, and if I add up the time I actually worked on LEDWARS, it would come to as little as five months — not bad for a solo programmer. But I could never have pulled this off if I hadn't set myself up to reuse a significant amount of LMK code in LEDWARS.

Problems Posed by LMK

A major recurring challenge during the development of LMK was finding ways to compress data more and more, and access it at increasingly faster rates. There's an old rule that states that if you spend a lot of time developing the compression scheme, the decompression step typically becomes faster. This rule of thumb proved to be true in the case of LMK.

Because LMK's game world can be modified during a game (for example, players can move objects), these changes

More Pathfinding Tricks

In my previous *Game Developer* article, which talked about pathfinding in real time, I talked about the problems that occur when there is no path. I suggested using the bidirectional A* path. An optimization that I didn't mention and that I used in LEDWARS is precalculating impossible paths. This technique is fairly simple and can augment your game speed tremendously. The game speed really benefits when you have a computer opponent that also needs to calculate paths. The opponent is more likely to try impossible paths, and if he keeps on trying, your game

speed will deteriorate quickly.

The idea is to create an extra layer that is the same size as the search space in which your search algorithm operates. This extra layer divides the map in several areas, so that every node within that area is accessible from another node within that area. This implies that there is no path between two nodes from different areas. A map with a lot of islands in it, therefore, contains a lot of areas, whereas a map that isn't complicated contains few areas. The following algorithm shows how to create such a layer.

Algorithm

1. Take a node that hasn't been designated to an area yet, and create a new area for it.
2. Do a breadth-first search from that node, assigning all nodes that can be reached from the root node to the last created area.
3. Repeat from Step 1 until there are no more nodes that haven't been assigned to an area

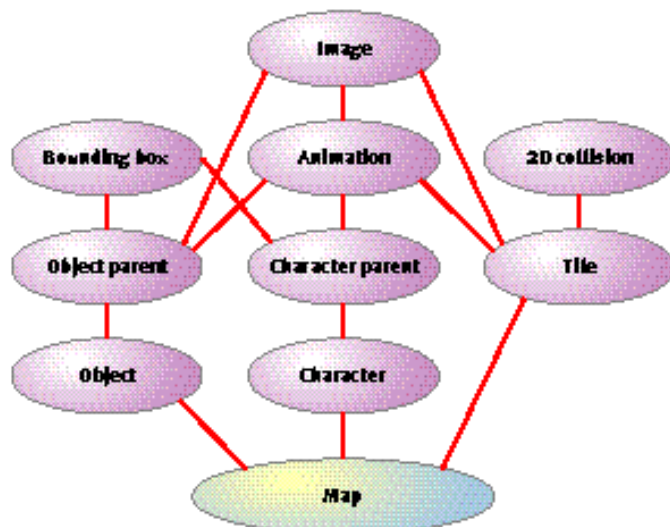
Once this extra layer has been created, all you need to do to determine whether a path is possible or not is to check whether the start and destination lie in the same area. Simple isn't it?

need to be written back into the compressed world. However, there are tens of thousands of objects in LMK, and each has its own set of variable data. This makes inserting objects into the map almost impossible. Insert operations in a block of information such as a map usually require that you move a large part of

that block. Since most of the map exists on disk, and not in memory, I can't move large parts of that block. To solve this problem, I created two maps: a highly compressed map of the original world, and a map that tracks the changes that the players make.

The first map is changed when an

FIGURE 5. Object relations for LMK.



The Bug Hunt

Even with the messaging system in place, the world organization set up, the structure of the AI taken care of, and a way to cope with numerous units, I was still late in delivering LEDWARS to my publisher. How could that happen? Given the explanation of data structures, it all sounds fairly easy. However, in the end, it was the AI and the multiplayer sections of the game that held up the game. My data structure design made it simple to change the AI. First it was too easy, then it was too hard. Ultimately I just put in a slider that you could change in real time to adjust the difficulty level.

Synchronizing the multiplayer system was also a horror. I spent several weeks trying to establish what was wrong with it. It's something I'll share with you so that you don't make the same mistakes.

The obvious flaws were a result of references to something the player did on the local machine that somehow affected the game flow. If a player clicks on a unit, a sample audio clip is picked randomly from a series and played. It was coded like this:

```
PlaySample(UnitSamplesList[FIGHTSOUNDS]
[u->parent][rand()%MAXFIGHTSOUNDS])
```

This desynchronized the random seed. Games often use random numbers to generate random behavior. In a system where it is important to have code that is determinant, it's important that the same random numbers are generated at the same time on all machines. Random numbers are generated using a formula that depends on a seed. The seed value determines which will be the next number in the series of random numbers. If you call `rand()` on one machine and not on the other, then both machines cannot be expected to generate the same random number.

A more serious bug was related to static variables. When we tested LEDWARS, everything was fine because everybody copied the game from a server. The static variables were all initialized to the same value. When someone played a single-player game first, and then joined a multiplayer game, all hell broke loose. What made this bug tricky to isolate was that its effects were seen hours after testers had begun playing; the static variable

was defined in a function that wasn't called on a regular basis.

LEDWARS' worst bug had nothing to do with the game play. After a while in multiplayer mode, various players' games were getting unsynchronized. Some unit-specific data was different on one machine than on the other machines. The game was designed in such a way that all events were completely deterministic. There was no function in the entire game that could affect the game state as a result of a local dependency. I checked to see if any functions could be generating unpredictable values. They weren't. The moment the game became unsynchronized, numerous debug reports were generated. The desynchronizing problems typically appeared after playing the game for a long time, but we couldn't isolate the problems. Even replaying the game using recorded macros couldn't reproduce the synchronization problem. We were stumped.

I'm amazed I actually discovered the cause of the problem. The game's options screen, where players could adjust settings for difficulty, volume, game speed, and so on, was 482 pixels high, instead of 480 as I presumed. Launching this screen corrupted the heap slightly, which much later generated erratic behavior on that player's machine.

When two machines in a multiplayer game under the peer-to-peer system don't have synchronized game states, check the following:

- Is there something that is generated locally that affects the game states? The random seed problem is an example of this.
- Are the default values different on different machines? Think of the problem that I had with a static variable, but also think of global variables that aren't reinitialized each time you start a game.
- Is the output of your functions machine independent? An 8MB machine has more memory than a 16MB machine, so the 8MB machine might disable some functionality in your game which is available on the 16MB machine.

If the answer to any of these questions is yes, you probably have a situation in which the memory is corrupted.

operation doesn't require insertion or deletion. For instance, lighting a torch is simply a matter of setting a bit in the compressed map. Moving a glass is trickier. A bit it is set in the first map indicating that the entry occupied by the glass is now free. Once the glass is put down again, the system checks if there is a free slot in the highly compressed map. If not, it puts the glass in the second map. This is an adequate solution, since the total number of objects moved by the players is much lower than the number of objects that they actually use. And as you can see, using objects usually changes data that is easily written back in the original map.

The client/server architecture greatly simplified how data is handled. Except for the objects, nothing needs to be synchronized, since the server handles most of this work. In LEDWARS, it was the peer-to-peer unit synchronization that caused a lot of headaches, and it would have been even worse in LMK if the characters used a similar system. The only thing I needed to bother with was designing the message structures so that a lot could be said in as little information as possible. My lesson? When given a choice, opt for a client/server architecture (see Figures 6 and 7).

Object Referencing

Objects (for example, tiles and units) reference each other using direct indexing. This is fastest and most convenient for debugging and reading/writing the structures to disk. For instance, in LEDWARS, a tile group contains a series of indices that refer to a global array of tiles. A tile collection has a series of indices that refer to a global list of tile groups. A unit parent references a tile collection using an index that directly references a global list of tile collections. A unit references a global list of unit parents again using a direct index. The same goes for buildings. The problem with this referencing system is that sometimes you get the impression that your data is far away. It may be a good idea to store some redundant data (particularly display code) within your classes to allow quicker referencing. For instance, if a unit is made up of several

FIGURE 6. LEDWARS uses peer to peer communication.

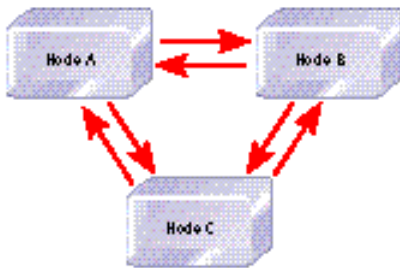
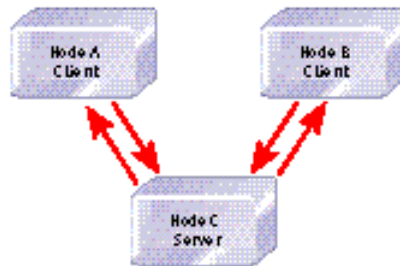


FIGURE 7. LMK uses client/server communication.



tiles, every time you need to draw a tile, you have to traverse the entire referencing chain, which costs you some speed.

48

Keep It Simple, Efficient

Although my team has been working on LMK for quite some time now, we still have another six months of development ahead of us. Throughout the project, we've tried to develop

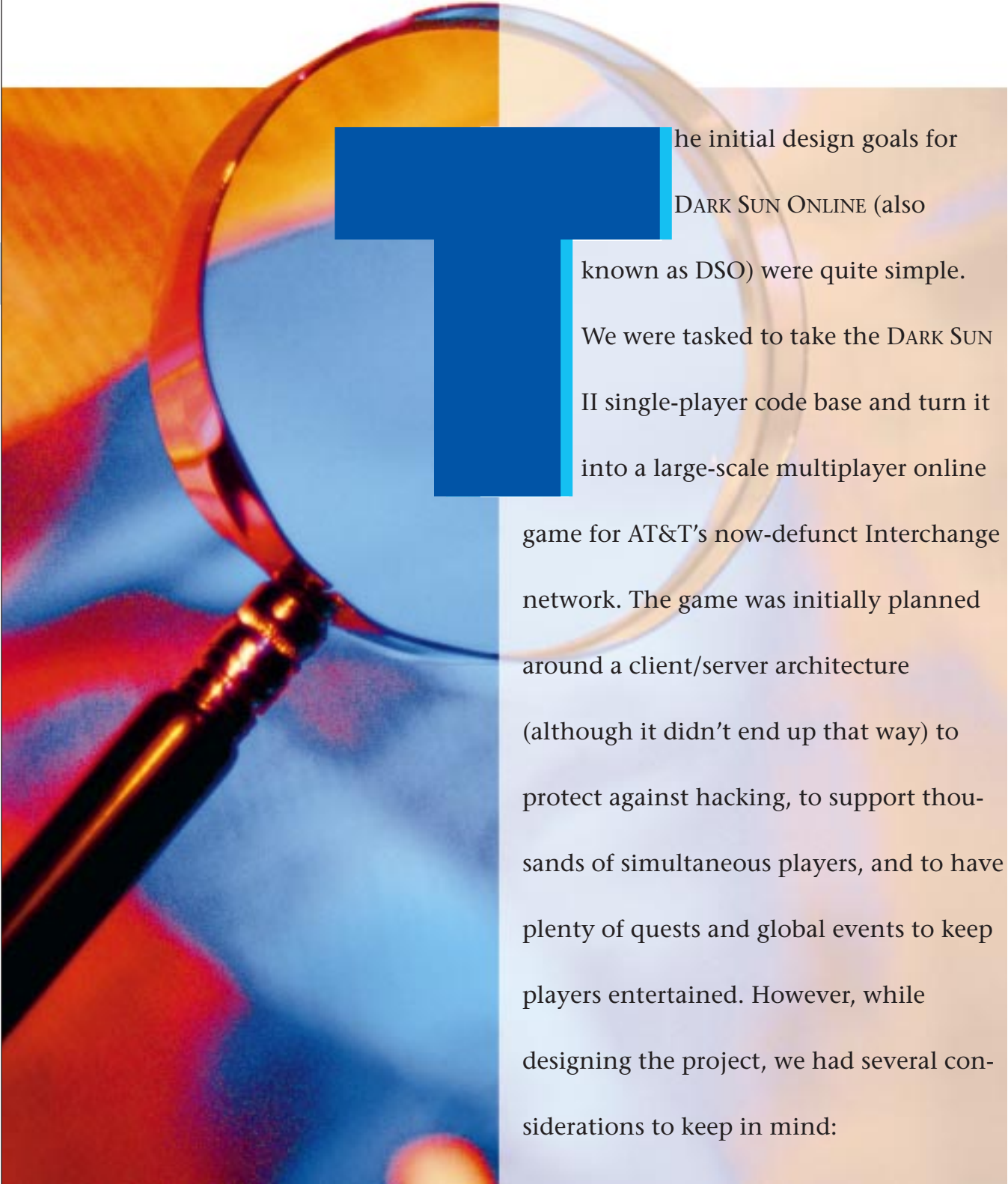
engines and editors that will be reusable in future games. Using database technology wherever possible was one means to this end. Database tools support multi-user environments and allow more than one editor to work on the same section of data. Database files can also easily be compiled into a game format, and it's easy to edit them using available tools.

What it all boils down to is reusability. Reusability reduces development

time for future projects and provides some other advantages as well. You are forced to consider how you structure data; the more complex your data gets, the lower your chances of ever reusing it. Another advantage of reusability is that it makes the task of programming a multiplayer game easier, since a componentized architecture lets you substitute multiplayer modules where necessary. Finally, the number of bugs is reduced, since components can be tested individually. The challenge to the developer becomes one of allowing smooth interaction between the components. Then again, if it all were so easy, everybody would be doing it. ■

Since Swen started coding, the stock price of Coca Cola went up. You can reach him at lar@larian.com. His company's web site, which contains more information about LEDWARS and LMK, is at www.larian.com.

DARK SUN ONLINE: CRIMSON SANDS



The initial design goals for DARK SUN ONLINE (also known as DSO) were quite simple. We were tasked to take the DARK SUN II single-player code base and turn it into a large-scale multiplayer online game for AT&T's now-defunct Interchange network. The game was initially planned around a client/server architecture (although it didn't end up that way) to protect against hacking, to support thousands of simultaneous players, and to have plenty of quests and global events to keep players entertained. However, while designing the project, we had several considerations to keep in mind:



First, our resources were quite limited, especially when it came to art and audio. We had one part-time artist available to create the few odd objects that we absolutely had to have; otherwise, we had to make do with art we could reuse and steal from other internal projects. We had the same limitations with sound and music resources and often had to go to extraordinary lengths to find quality material.

seen the success of NEVERWINTER NIGHTS on America OnLine, the company wished to fund the development of a sequel based on the DARK SUN engine for its exclusive online use.



Our first challenges became apparent in October of that year as the contract was signed and work began. Due to some overly aggressive estimates as to the ease of porting DARK SUN II to an online environment, the project was basically underbid from the very beginning. This had many ramifications, beginning with a lack of art and sound resources to draw upon, and continuing with difficulties keeping external team members' morale up.

Thanks to these resource constraints, we had to go to great lengths to get the material needed to create the game. To begin with, we raided the source code archives and reused

PORTING A STAND-ALONE, DOS-BASED RPG TO A 32-BIT WINDOWS-BASED ONLINE NETWORK CAN BE...CHALLENGING. HERE'S WHAT HAPPENED TO SSI ALONG THE WAY.

by André Vrianaud

Second, we knew we had to use the existing code base and couldn't rewrite the game from scratch. This obviously tied us down with a great deal of baggage, not the least of which was that we were using a dated game engine, both graphically and in terms of the game's code. Because of this we chose not to focus on the graphical aspects of the game, which would have been futile, but dedicated all of our resources toward creating a multiplayer game with strong game play elements.

Finally, we had to coordinate all of our efforts with an external programming group and contract art group. As you'll see, there were definite challenges with meshing the internal game design and the external programmers. In addition, we had to go through many iterations of art creation for some of the new characters and monsters. This proved to be an entertaining experience in its own right.

Where It All Began

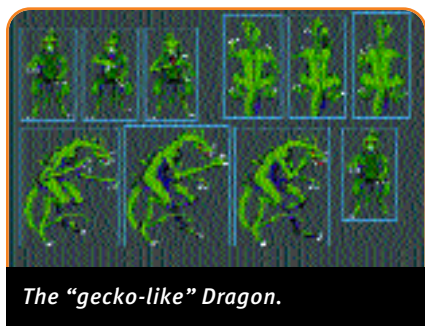
The origins of DARK SUN ONLINE can be traced back to the summer of 1994 when representatives of AT&T's Interchange network approached SSI to do a new ADVANCED DUNGEONS AND DRAGONS online game. At that time, AT&T was in the process of planning its own proprietary online service, and believed that having a strong game presence would be of great help in launching the network. Having

art from both DARK SUN I and DARK SUN II. This turned out to be more difficult than it sounds, since there was a slight perspective shift from the first game to the second, and much of the early art simply didn't look right in the new DARK SUN II perspective. We also borrowed a great deal of art from an SSI title called AL-QADIM, which had an Arabian Nights theme. Although much of the art was unusable for the same perspective reasons, some fit perfectly. Sand is sand, after all, no matter what angle you view it from!



For the entire duration of the project, we only had a half-time artist to do some miscellaneous objects, interim cut-scenes, and cinematics. Since there was always a huge crunch going on with other internal projects, we also had to deal with losing that artist at the oddest (and most frustrating) times — especially since there





The "gecko-like" Dragon.

was enough work to keep several artists occupied for months. One of the biggest effects that this situation had upon the original game's design was the loss of all of the game's cinematics, apart from the introduction. (Ironically, even this introductory cinematic was only available for a short time in the DARK SUN ONLINE 1.0 retail version. Later versions of the game were never re-released at the retail level, and the introductory cinematic scene was too large to be included in the downloadable versions.)

Additional art work that needed to be accomplished for this project included the customization of hundreds of player icons, some perspective tweaks on DARK SUN I monsters, and the creation of several new monsters. Since we effectively had no internal resources to create this art, we turned to an external art house.

The character portraits went relatively well, with only a few redos on dental-floss bikini bottoms and overly-endowed female characters. The Dark Sun I monsters were also tweaked with little problem. Unfortunately, the new monsters turned out to be far more challenging for the external artists. Two monsters in particular — the Dragon and the Nightmare Beast — were somewhat less than fearsome. As you can see from the sample art, the Dragon (even *after* some rework) looked like a gecko, and the Nightmare Beast... well, the Nightmare Beast just looked like Barney. Perhaps luckily, the Dragon encounter ended up being cut from the game, and since we never could get Barney looking right, his appearance was cut, too. (There were those who thought keeping Barney in could be quite a cathartic experience to certain twisted people, but fortunately saner minds prevailed.)

We did much better as far as sound effects went. Digging in the archives, we

were able to draw upon the sound effects from Dark Sun I and II, which pretty much covered most of what we needed. However, we were still a little short in some areas and cast about trying to find a few more additional effects. Lo-and-behold, the sound department coincidentally sent out a large archive of .WAV files from the soon-to-be released THUNDERSCAPE title for use as Windows event sounds. Much to the chagrin of some on the THUNDERSCAPE team, those effects quickly found their way into DSO.



Much of the art from DSO was reused from earlier games

Using Contracted Developers

Beyond all of the internal resource issues, DARK SUN ONLINE was also an interesting project due to its use of external programming talent for the online coding and porting of the game. At the time, SSI had no internal online programming talent available to code a multiplayer version of DSO. So the online coding was subcontracted out to an external programming group. Now, working with external groups is pretty common in the industry, and normally isn't that big of a problem. Unfortunately, due to some political and financial issues, the contract was finally signed (over a few objections) with the external group getting paid on a strict milestone basis, with no royalty points or interest in the finished title. As you can



"Barney," the Nightmare Beast.

imagine, this became a bad morale issue for the group.

This morale problem reared its ugly head quite quickly as the external programmers and internal scriptors began to interact. Earlier DARK SUN games (other than the art work) were created as a joint effort between the programmers and the scriptors (also known as the designers). The programmers would work on the game engine, adding features such as spells, new monster effects, and new GPL (Game Programming Language) commands for the scriptors' use. Meanwhile, the scriptors would use GPL to code the actual adventure parts of the game — the events, quests, and NPCs (nonplayer characters). The key to this arrangement working harmoniously was that most of the GPL-interpreting code (in the engine itself) was already done, allowing the scriptors to work relatively autonomously. Unfortunately, this wasn't the case with DSO.

The big problem with this project was that the external programmers had to take a code base for a game that was designed as a single-player game and make it a multiplayer game. Unfortunately, the original programmers (on DARK SUN I AND II) made assumptions about the way certain GPL commands would work, in order to save time on coding error-checking routines. For example, the original programmers never had to deal with the possibility of two different people trying to talk to the same NPC at the same time; it simply wasn't possible in a single-player game. However, this situation is extremely common in the online version. The result of this was that the external programming group had to recode the way GPL worked — many, many times as the game's development progressed and we learned more about how the engine would really work.

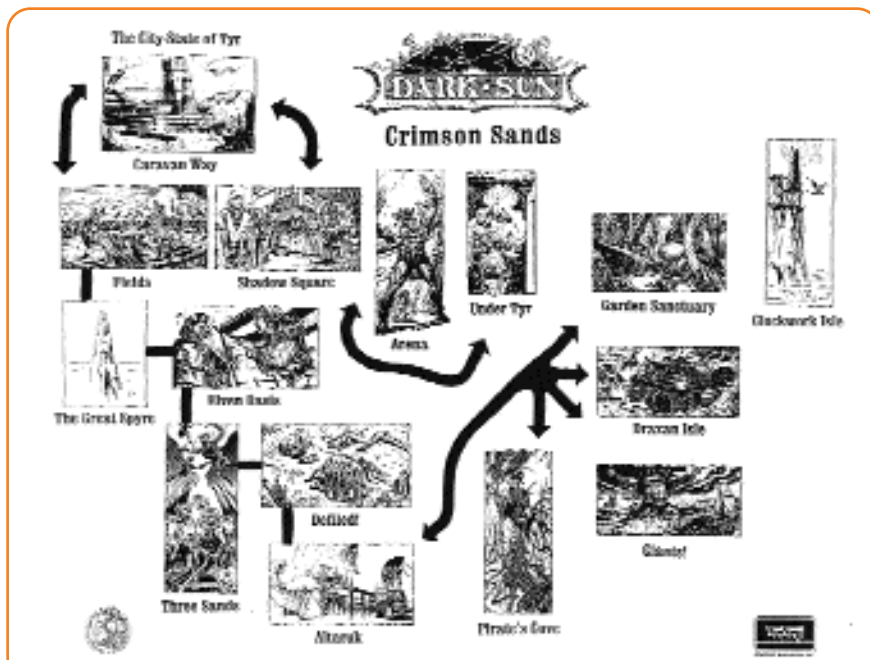
Rich Donnelly, the lead scriptor for DARK SUN ONLINE says, "Oftentimes, I would write script for the engine, and the engine would change halfway through, in such a manner that the script would no longer function. Even worse, there were times when crucial information was lost in the associating chain, such as new GPL functions or commands. This caused a lot of strife and recoding on everyone's part."

To sum up, since the external programming group had no buy-in or incentive to make the project the best it could be, they generally took the easier path whenever possible. The most grievous example of this was the networking change from the planned client/server architecture to a peer-to-peer system. Due to low morale and compensation, the external group insisted on coding the networking architecture as a peer-to-peer network, using the local clients to run most of the game logic. Although we raised the red flag multiple times about problems inherent with this scheme, our team was overruled in the interests of keeping costs low. To be fair, the peer-to-peer coding was done competently for what it was. Unfortunately, though, this type of networking code wasn't the best for the game at large, and we had to deal with the hacking issues that it raised for the life of the project.

The Debugging Process

A few months passed, and eventually the programming group got the engine to the point where the scriptors could do some GPL coding without having to restart every month. At this point, the scriptors dug in and started coding the underlying global routines for the game, and the external programmers started modifying the game engine to work over an IPX network, which would allow us to test and develop internally. Within a few weeks, we had that IPX-capable version of the engine to work with — which raised new issues.

Chronologically, this was soon after DOOM was released, and if you'll recall, the early versions of DOOM had problems with packet flooding, which brought down quite a few corporate networks. This was quickly fixed in



Sysop tools were added that allowed the DSO world to expand beyond its original size and let players discover new challenges.

subsequent releases, but the internal management at SSI was still a little gun-shy, and asked us to set up a completely separate network for our development work. This in turn meant we needed a UNIX box to run the game's server code, which meant we needed to set up and configure a Linux box (Linux is a free variant of UNIX) to act as the host. To make a long story short, configuring a UNIX machine isn't an easy task, and by the time we got the hardware, built the network, and configured the host, we had lost a few weeks of development time.

The greatest hit that our original timeline took resulted from the initial plan to run DARK SUN ONLINE as a DOS application in a DOS box, but communicate with a Windows-based TCP/IP stack. AT&T's Interchange online service was a Windows 3.1 application and ran over a proprietary TCP/IP stack that AT&T had licensed. Since the original DARK SUN games were DOS applications, and since Windows 95 hadn't yet penetrated the market sufficiently, it was thought best to keep DARK SUN ONLINE as a DOS application. We would depend on a "shim" of sorts to allow the game to communicate with the Windows protocol stack. We spent a couple of painful months trying to get this going, and eventually had something that worked, but not well. The shim would occasionally lock

up for seemingly no reason, and we were unable to get the source code for the protocol stack to try to fix the bug. Eventually, we scrapped that effort and ported DSO to a true Windows application. In retrospect, we probably should have bit the bullet early on and moved the game to a Win32 application from the start.

After this initial setback, work progressed for a few months without any particular disasters to note. Art from the external group was slowly coming together, and we resolved most of our sound and music issues. The external programming group was hitting its milestones, and all in all, everything was quiet. Yes, as many of you just guessed, too quiet.

In early November of 1995, I received a call from the head of Interchange's game group. It turned out that AT&T had pulled one of their classic turn-arounds and decided that it didn't actually want to be in the online network business after all. In short, AT&T's Interchange network was canceled, along with all of the projects being funded for it — including DARK SUN ONLINE.

At first, it didn't look too bad. Although we didn't have a network to launch the game on, we had gotten a great deal of the project funded, and through a nice quirk in the contract, all



rights to the game reverted back to SSI. This allowed us to shop the title around to different networks and possibly negotiate a better royalty deal. Unfortunately, this process took several months, and since SSI's internal management wasn't confident that they'd be able to find somewhere to place the title, the project was placed on a back burner. A couple of the scriptors and I continued development, along with the external programmers, but little other work was done. However, a few months later, it was decided that TEN was the appropriate network for the title, and work began again in earnest in January of 1996 — with one other little bump.

In mid-January, one of the three internal scriptors decided to move on to new opportunities. This normally wouldn't have been that big a deal — things were moving along well, and the two remaining scriptors would simply have had to pick up the remaining work and take care of it in the extra months we'd allocate for them. However, the scriptor that left was responsible for a great deal of the global game code upon which the game, along with much of the other scriptors' work, depended. Sadly enough, it turned out that much of that work had either not been done, or had been done so poorly that the code couldn't be depended upon. It fell to Rich Donnelly to pick up the pieces.

"It was quite horrific for me to find that the game was missing some serious pieces of code. These were pieces that were vital to an online environment, such as global region transfer code that handled moving entire par-

ties instead of individuals, or monster generation routines for general combat encounters that could incorporate several different players in the same combat. Suddenly, I found myself with little time to correct these problems. I worked heavily for about a month, and managed to finish getting everything implemented and working."

The Beta Test

The extra months added to our schedule after AT&T folder their service were quite useful, as they gave us some extra time to prepare for the external beta test. Although we had a core group of internal testers on the project, we knew that we'd need to run a large-scale external test to really shake out all of the bugs. To that end, we prepared a web page that gave instructions on how to sign-up for the beta test — and were quickly inundated with responses.

We originally planned to solicit beta testers and mail out a CD-ROM to everyone who responded. Unfortunately, so many people responded that we couldn't follow through and were only able to send out around 500 discs; however, the rest of the testers were able to download a 30MB PKZipped version of the game.

The beta period in particular proved to be quite an educational experience, and we definitely learned some lessons from it. In fact, it's probably worth taking a bit of space here to list some of them.

LESSON ONE. You can never allocate too much time for beta testing and debugging an online title. The sheer number of bugs found by the external testers was absolutely amazing — and the potential for each of those bugs to cause havoc was greatly magnified by the interactive nature of the game. Allocate significantly more time than you first think for your testing period — and double it. Seriously.

LESSON TWO. Be sure to have robust facilities to deal with the flood of bugs. I highly recommend setting up some method for external testers to enter their bugs on a web page, and a method to import those bugs into whatever bug database you use. We had the bug submission page, but no way of automatically moving those

reports into our database. In fact, due to the kludged nature of our bug-tracking system, we ended up having each and every bug sent to my office e-mail account. I would then log in every morning and run a mailbox filter, forwarding all of the bugs to my lead tester to sort, compile, and enter into our internal bug system. I don't think he's forgiven me to this day. **LESSON**

THREE. Have a FAQ. Period. Every morning I would spend hours reading and answering hundreds of messages. Some were suggestions, some were flames, but most were simple questions — and we quickly realized that we were seeing a great deal of them over and over. However, once I got the FAQ written and published, the flood relented... a little. Beyond that, though, the FAQ turned out to be a great promotional and educational tool; it even staved off our marketing department a few times by proactively giving them whatever information they were looking for. Really, now — can you ask for a better reason to do a FAQ than that?

LESSON FOUR. Don't hype the product until it's ready. We purposely kept quiet about the title until very late in the project and were thus mostly able to handle peoples' expectations. (I say "mostly" because there are some people you'll never be able to satisfy.) We wince now when we see the hype that's been built around ULTIMA ONLINE. Although UO is likely to be an impressive and groundbreaking product once it's finished, it's also unlikely to ever satisfy the heightened expectations that have been built up around it.

LESSON FIVE. Make it extremely clear that it's a beta-test. Then do it again. And again... and yes, yet again. Even so, I guarantee you people will threaten, flame, and otherwise get extremely mad at you when changes are made. As Donnelly says, "Players are not concerned with the fact that the game may not be in its final state — if you change anything, there will be com-



plaints, regardless. During our beta test, we altered the way the environment worked several times and wiped the host quite often to insure that the test was started afresh, with no corruption. Naturally, players screamed at us every time this happened. In addition, watch what you show the players during testing periods, as people aren't thinking about what the project will look like in the future; they care about what it looks like now. I recall that ULTIMA ONLINE got quite a negative reputation initially when Origin showed their alpha, as players immediately assumed that was what the game was going to be like. Beta testing includes more marketing than most people realize."

LESSON SIX. Hackers will exploit the slightest loophole, and it's often that exploitation that ruins the game for all of the other players. DSO was particularly hackable due to its odd peer-to-peer architecture — an unfortunate legacy of the external programming group doing things the easier way for them. Again, if you're doing to do a large-scale RPG on the 'net, don't consider building it around anything but a client/server architecture — and make sure the server is the arbitrator of all key game logic.

After a long and painful beta period, DARK SUN ONLINE was finally launched live on TEN — and as dictated by Murphy, the inevitable bugs popped up. DSO version 1.1 was released a few months later to address most of those bugs, and DSO version 1.5 came along a few months later as a game expansion. Discussions are underway as to the possibility of doing a DSO 2.0 — and as the player base continues to grow, I'm sure we'll see the game expand and evolve even more in the future.

Modifications Along the Way

DARK SUN ONLINE differed from its original design goals in a number of ways. First, the peer-to-peer architecture we ended up with limited us to hundreds of simultaneous players instead of the planned thousands. That same peer-to-peer architecture also made the game extremely hackable, as much of the game logic was processed on the local machine instead of the host.

Second, a great deal of new sysop tools were added due to beta tester feedback. Those tools, in turn, helped keep the game viable by allowing staff members to run hand-made events while the scripted quests were being fixed. In addition, extra regions were added to give the players more room, and even more regions were added for version 1.5.



Third, due to the easily-hackable architecture of the game, players were able to cheat and escape routines to punish character death. Version 1.5 of DSO removed those penalties until we could move the game logic to the server — hopefully in a future version of the game.

Finally, all cinematics were cut except for the introduction for version 1.0. That introduction cinematic was also cut in post-1.0 versions, due to its being too large to realistically download. In addition, since the game was never re-released at the retail level after version 1.0, the Red Book music became unavailable. However, it's possible that the original DARK SUN I MIDI tracks might be reintroduced in future versions of the game.

Despite these changes, however, DARK SUN ONLINE stayed with most of its original design goals. The game play was our focus, not the graphics. Some of this was simply because of the original art that we were forced to use, but a great deal of it was due to beta testers' suggestions during the testing period.

Player interaction and communication was the focus of the entertainment, and not prescribed elements as in the original DARK SUN games. We also reused legacy art to great effect (albeit cheating like dogs the whole while to steal decent).

Finally, the online interface added a great deal of functionality, while remaining inspired by the original DARK SUN games and thus familiar to players' of those titles. The chat system

in particular was quite powerful and worked out well.

A Look Back at the Good, Bad, and...

I'd like to highlight a few things that went particularly well during the development of DARK SUN ONLINE... and a few things that didn't go so well. Some of them have already been touched on above, but there are a few others also worth mentioning.

WHAT WENT RIGHT?

1. A close feedback loop with the external testers proved to be incredibly helpful. Other than the obvious benefit of learning about bugs and interface problems, we also got a good deal of free publicity and goodwill. That goodwill was particularly useful when we made the inevitable screw-ups. In addition, it was beta-tester feedback that spurred development of the FAQ.
2. The chat interface turned out to be far more intuitive and user-friendly than we ever thought it might. Rich Donnelly says it best. "The chat interface is probably one of the most dynamic and user-friendly chat interfaces to be seen in any online role-playing game. In almost every case, there are multiple ways to do the same task, something that almost every user enjoys. As a designer, I have found that players adopt their own style of play, regardless of how the interface or game environment is designed. Having the ability to perform tasks in a variety of different ways allows the player to find the method of play that is most comfortable to them. Look at Windows 95, for example. Some people prefer to work from their Desktop, others like it nice and clean and prefer to use the Start menu and Explorer to find the items that they're seeking. It is this versatility that people desire, and including it in your product is essential."
3. The reuse of DARK SUN I and II, and AL-QADIM art assets proved to be an effective decision. Although many on our team would have liked to improve the existing art, or create a great deal of new art done, we were able to reuse most of the art from DS II, some from DS I, and a little from the external project AL-QADIM to good effect.



4. Our online role-playing paradigms were well thought out, and some are finding their way into other games. We learned that you can't allow a real, permanent character to die if a player was paying for it. We discouraged and punished people to keep them from cheating and skipping out of combat. DSO also allowed and encouraged player vs. player combat — an element we see being supported in more and more online role-playing games. Finally, we implemented the concept of instant communication across the online world, no matter where you were. Purists seem to disagree and dislike the lack of reality, but those purists also forget that chatting is the single most important community support tool you have. Nothing in your game design should ever discourage communication among players.

5. Although the basis of the game is player interaction, we found that having a strong random quest generator helped fill in the gaps. To quote Rich, "Having a system that can essentially generate quests on its own is something that definitely increases the entertainment value of the game. The quest generation system currently in the game is rudimentary at best, but it's definitely a solid model and a step in the right direction for what might be termed a 'story-telling engine.' As is the case with all developers, my only regret is that I didn't have the time to take this quest engine as far as we would have liked. However, the model as it stands is a serious piece of machinery, and something to be proud of."

WHAT WENT WRONG?

1. We tried to make a DOS application running in a DOS window communicate to a Windows 3.1 TCP/IP stack. Instead, we should have simply ported DSO to a Win32 application and developed the game with the Windows 95 TCP/IP stack — as ended up being done months later.
2. SSI's internal resources were severely limited for the project, which in turn made us far too dependent on external resources. On top of that, the contractors should have been better incented to provide quality work. In



- particular, the external programming group was competent, but not financially-motivated to do the job "right." In addition, the external art group needed a great deal of hand-holding, and at the end of the day we still didn't quite get everything we wanted.
3. One of the scriptors left in the middle of the project and didn't do a great deal of the work that was necessary for the game's underlying routines. We ended up having the two remaining scriptors doing the work of three, with one of them having to redo all of the key global routines.
 4. Not enough test time was allocated for debugging such a large-scale online game. This forced us to move extremely quickly which, in turn, frustrated players due to the speed and severity of the changes. Although many of these issues were unavoidable, having more time to prepare and a more flexible deadline would have allowed us to be more accommodating to testers' issues and frustrations.
 5. Probably the biggest issues we had during the development of DARK SUN ONLINE were the hacking problems. These problems came as a result of taking a stand-alone game

and turning it into a multiplayer game with a peer-to-peer networking system. Game logic then ran locally on the client, rather than on the host. Rich Donnelly has a little more insight on this, and explains it thusly:

"The game environment itself, having been converted from a stand-alone product to an online product, left the game with most of the logic running on the user side. The host itself merely transfers the information back and forth between all the players, keeping everyone in a huge loop of data sets. This causes some serious problems with hacking, as people could use editors on their local machines and change critical data. "Hacking thus became quite a nightmare for DARK SUN. Hacking a stand-alone product is no big deal — change the game all you like, nobody is going to complain. In an online environment, you are affecting not only yourself, but also everyone else playing the game. It's kind of like a movie theater; you purchase a ticket for one seat, but you don't buy the whole theater. When you begin screaming, the ushers escort you out, as you are disturbing others. It's the same for an online environment — it just takes one bad apple for everyone else to feel the effects."

And On the Seventh Day . . .

I hope this article has given you a few things to think about, and helps you in the development of your title. I'm always interested in discussing gaming and the online industry in general, so if you have any comments or questions, please feel free to drop me a note. ■

André Vrignaud has worked in the computer gaming industry since 1990. Most recently, he's worked at SSI and TEN, where he produced much of their online content and direction. He wishes thank everyone involved with DSO, with special thanks to the Lead Scriptor/Designer Richard Donnelly, TEN's RPG Producers Alex Beltramo and Don Hupp, and in particular, all of the thousands of external beta-testers. He can be reached at andre@null.net.





Goodbye For Now

W

hat an interesting industry. I enjoyed creating my last game, ENEMY NATIONS, more than any other project I've ever worked on. The game market is more "alive" than, say, operating systems, where whatever we (I used to work for Microsoft) released generally succeeded.

64

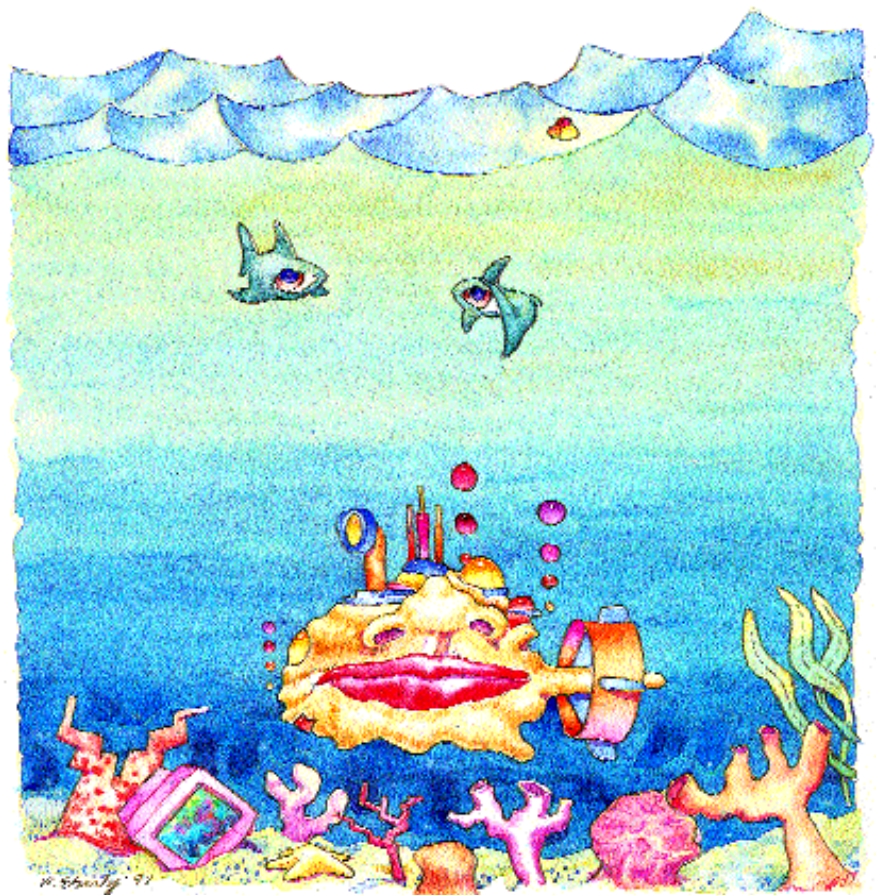
But the game industry, in spite of having its highest grossing year ever, is in serious trouble. Virtually every company is hemorrhaging money. You can't make a profit in a market like this. Losing money while chasing market share is only for companies with extensive cash reserves.

Let's travel down the path of confusion, incompetence, ignorance, and greed that has brought the game industry to this point. First, retail is broken. Stores purchase from distributors and distributors purchase from publishers, all of whom have no intention of paying. The effective discounts achieved by finally settling accounts for less is a giant tax imposed on publishers, and through them, on developers.

Management at most computer store chains is so inept that publishers request letters from the chain's corporate headquarters stating what programs the publisher has paid for. The publishers then send their channel marketing reps to the stores, letter in hand, to get the programs that they paid for implemented. In a word, chains are incapable of implementing the sales programs that they sell to the publishers in their stores.

The stores also have no idea how to stock titles. I tried to buy DEADLOCK a month after it came out, but not a single store in Boulder, Colo., carried it, had carried it, or knew that it existed. It was a top-selling game at the time. No matter how good the games are, if they aren't on the shelves, people won't buy them.

Second, many game publishers are grossly incompetent. Viacom New



Media, GTE Interactive, and Phillips New Media are just some of the recent companies to be closed down after failing to produce, in some cases, a single successful title. Meanwhile, Virgin Interactive (according to Spelling's latest 10-K) is losing more than five million dollars per month. (I do want to note that there are a few well run game companies out there.)

Some publishers with which I interacted had employees who, in my opinion, were totally incompetent. I knew a test manager who had no idea what a test plan was. I knew a vice president of development who did not know how her company tested games. I knew a vice president of software who did not understand disk I/O optimization (although he thought he did).

Continued on page 63.

Continued from page 64.

Publishers don't understand quality titles when they are right in front of them. Viacom New Media dropped ENEMY NATIONS after looking at the final game. VNM is a company that received an average rating of 67% in *PC Gamer* last year for their games. *PC Gamer* gave ENEMY NATIONS an 86%. Yet even looking at the final game, VNM couldn't determine if it was better than the dreck they did decide to ship.

When VNM returned ENEMY NATIONS, we took it to every publisher we could find. Every single one turned it down. About four months later, *PC Games* reviewed it and gave it an A rating. Then we received calls from many of those same publishers, who were now suddenly interested. The difference was that a single person — the magazine reviewer — had looked at the game and liked it.

If publishers can't even trust their own judgment, if they turn down a quality game in the hottest genre out there, then what are the chances of their picking up any game brought to them? More importantly, if every publisher turns down a finished game with the ratings we received, what are the odds that they are accepting quality titles?

The third problem, though, is the biggest. And it's something we all keep quiet about while pretending it doesn't exist. Many, many game programmers are not qualified for the job. I'm not saying they're stupid (at least not all). I'm saying they're not qualified.

Most game programmers are self-taught or come straight from college. They work in small groups, usually with the same people year after year. Because this industry hasn't demanded exceptional programming talent, most game programmers get their experience in small, noncompetitive, incestuous groups — not an environment that pushes people to constantly strive to do their best. It is not an environment that teaches people new approaches and methods, either. (That said, there are numerous exceptions. I have been in both Chris Robert's and John Miles' code, and it contains beautifully crafted assembly code.)

When developers have to "discover" that you can process **duords** as fast as bytes, when they don't understand the concepts behind basic tools like a pro-

filer, when they chew up clock cycles reading the joystick eight times per frame, then they are not capable of writing good, solid code. All they can aspire to is adequate code.

The heads of several game publishers have told me that they have no interest in hiring very experienced people, because it's cheaper to hire programmers fresh out of college and pay them almost nothing. (And with rare exceptions, these are people who are not getting offers from companies such as Microsoft and Netscape.)

The mass market that we all desperately want to create will not accept adequate programming jobs. They demand the same level of reliability, completeness, functionality, and ease-of-use that they get from their productivity applications. And when they don't get it with one game, they stop buying games for good.

So what are the implications? Very few developers will get rich in the game industry over the next couple of years. There just isn't enough profit to spread around. At the same time, with very rare exceptions, game programmers who can write excellent code learned their craft in non-game backgrounds. When gaming does demand (and is willing to pay for) exceptional programmers, they will be hired from non-game companies.

Programmers are in incredible demand across many different industries. Microsoft has two billion dollars allocated to R&D that they can't spend because they can't find enough qualified programmers. At the other end of the size spectrum, my current employer, Fusion MicroMedia, is also desperate for qualified programmers. There are tons of jobs out there for those who want them.

To anyone who is contemplating working in game development, I offer the following advice. First, work three or four years for a company outside of the game industry. Although games are content and therefore more interesting than developing, say, Internet applications, working with a group of very smart and qualified people more than makes up for it. Besides, coworkers and management in other industries are usually extremely well qualified — they have to be to succeed. After that experience, you will be a much better developer, you will have made more money,

and you'll have gained a lot more experience. You'll be capable of creating much better programs in a much shorter time. And most likely, by that time, the game industry will be populated mostly by competently run companies.

(As a side note, everyone always complains about Microsoft winning new markets by using some kind of special advantage. It's a lot simpler than that. Look at the problems that I've listed in this column. Microsoft doesn't have those problems. Microsoft will come to dominate the game industry not because of any dirty tricks, but because that company is run and staffed by competent, intelligent people. That alone gives them a significant competitive advantage over virtually every other game company out there.)

I'm really glad I had the chance to create ENEMY NATIONS. It was a rare experience to be the producer, designer, tech lead, co-art director, and music and sound effects director, as well as be responsible for a good chunk of the programming. There is absolutely nothing like creating content. Even after everything that happened to me during the course of the game's development, I'm glad I did it. I learned a great deal and enjoyed it for the most part. But I'm also glad to be out and working again in an industry that demands excellence.

To those of you that choose to stay in the game industry, I say "thank you." I most certainly want to keep playing new games over the next couple of years.

However, I'm sure the industry is going to face some changes in the future. If nothing else, competitive pressure from Microsoft will significantly improve the titles coming from those companies that survive. ■

Dave Thielen is an internationally recognized expert at practically everything. When not being flown to Washington, D.C., to provide personal advice to the president, he is working with Steven Spielberg on his latest film. He has just completed his latest book, More Accurate Than God, Quotations from Bill Gates.

