gd

GAME DEVELOPER MAGAZINE

DECEMBER 1999

UBM Tech

Down With Global Homogenization!

I think most of us in this industry are pretty happy about the fact that interactive electronic entertainment has rapidly become a widespread form of entertainment around the world. And in case you didn't notice, game development itself has spread around the globe with similar speed. This fact was made abundantly clear to me last year when entries for the GDC's Independent Games Festival poured in from far-flung countries like Poland and Pakistan. (That was a bit of a mind-blower.) It really makes you realize that there's much more to this industry than meets the eye at E3, ECTS, and the Tokyo Game Show.

What's becoming increasingly clear to me is that many overseas game developers want to come to America. I know this firsthand, because I regularly get asked to write letters of reference for foreign game developers trying to make their way through the U.S. Immigration and Naturalization Service bureaucracy. While my experience is just anecdotal, I've spoken to enough game executives around the industry to know that many American companies actively recruit overseas game development talent.

It's probably not too tough to entice a foreign developer over here. It's no secret that American companies pay game developers more than those in most other countries (and that probably holds true for most professions). A salary survey conducted last spring by the Miller Freeman Game Group (which this magazine is affiliated with) and Market Perspectives revealed that the average total salary — including any sort of cash bonus — for an American game programmer in the U.S. was $49,991, and the median total salary for a game programmer was $50,000. Compare that to a Polish company that we recently talked to, which pays its staff programmer the equivalent of about five hundred U.S. dollars per month. Though the cost of living in the U.S. is substantially higher, that magnitude of a discrepancy lures many foreign game developers away from their native lands.

As an American, however, I must admit to having reservations about an influx of foreign talent. It's not that I think American jobs will be stolen by immigrants, nor that I adhere to isolationist beliefs. On the contrary, I say the more the merrier here in the U.S. What I fear is the result of a slow, steady exodus of game developers from countries whose game industries are just beginning to form. I don't think that's good for the countries in question, nor for their burgeoning communities of game developers. Sometimes all it takes is a few key people leaving a team to kill a game, and in some countries, the development of a single commercial game is a real feat.

I also feel (and I think many would agree) that our industry needs to explore more game designs, and I fear anything that will homogenize game development. Cultural differences between countries make many titles extremely entertaining, simply due to their (for lack of a better term) exotic design. When I first saw PARAPPA and DANCE DANCE REVOLUTION, I knew they weren't developed in America. There is something distinctly Japanese about them which I really enjoy. It would be a pity to lose some of that diversity in favor of Yet Another FPS.

In the newspaper this morning I read that French chefs staged a protest in Paris, in part to voice their anger against America's growing economic and marketing muscle overseas — in this case, McDonald's was the object of their scorn. The French want no more of our fast food, music, and movies. They probably still hold Euro Disney against us. And while I used to write these complaints off as poor sportsmanship in the arena of international business, I'm coming around to see that it's important to grow our industry outside the clutches of Uncle Sam.

Sigh. Maybe a croissant will cheer me up. ■

ON THE FRONT LINE OF GAME INNOVATION
Game DEVELOPER

600 Harrison Street, San Francisco, CA 94107
t: 415.905.2200 f: 415.905.2228 w: www.gdmag.com

Publisher
Cynthia A. Blair *cblair@mfi.com*

EDITORIAL

Editorial Director
Alex Dunne *adunne@sirius.com*
Managing Editor
Kimberley Van Hooser *kvanhoos@sirius.com*
Departments Editor
Jennifer Olsen *jolsen@sirius.com*
Art Director
Laura Pool *lpool@mfi.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Jeff Lander *jeffl@darwin3d.com*
Paul Steed *psteed@idsoftware.com*
Omid Rahmat *omid@compuserve.com*
Advisory Board
Hal Barwood LucasArts
Noah Falstein The Inspiracy
Brian Hook Verant Interactive
Susan Lee-Merrow Lucas Learning
Mark Miller Harmonix
Dan Teven Teven Consulting
Rob Wyatt DreamWorks Interactive

ADVERTISING SALES

National Sales Manager
Jennifer Orvik e: *jorvik@mfi.com* t: 415.905.2156
Publisher Relations Manager
Ayrien Houchin e: *ahouchin@mfi.com* t: 415.905.2788
Account Executive, Eastern Region
Afton Thatcher e: *athatcher@mfi.com* t: 415.905.2323
Account Executive, Western Region
Darrielle Sadle e: *dsadle@mfi.com* t: 415.905.2182
Account Executive, Northern California
Dan Nopar e: *nopar@mfgame.com* t: 415.356.3406
Account Representative, Silicon Valley
Mike Colligan e: *mike@mfgame.com* t: 415.356.3406

ADVERTISING PRODUCTION

Senior Vice President/Production Andrew A. Mickus
Advertising Production Coordinator Dave Perrotti
Reprints Stella Valdez t: 916.983.6971

MILLER FREEMAN GAME GROUP MARKETING

Marketing Director Gabe Zichermann
MarCom Manager Susan McDonald
Junior MarCom Project Manager Beena Jacob

CIRCULATION

Vice President/Circulation Jerry M. Okabe
Assistant Circulation Director Sara DeCarlo
Circulation Manager Stephanie Blake
Assistant Circulation Manager Craig Diamantine
Circulation Assistant Kausha Jackson-Crain
Newsstand Analyst Joyce Gorsuch

INTERNATIONAL LICENSING INFORMATION

Robert J. Abramson and Associates Inc.
t: 914.723.4700 f: 914.723.4722
e: *abramson@prodigy.com*

Miller Freeman
A United News & Media publication

CEO/Miller Freeman Global Tony Tillin
Chairman/Miller Freeman Inc. Marshall W. Freeman
President & CEO Donald A. Pazour
Executive Vice President/CFO Ed Pinedo
Executive Vice Presidents Darrell Denny, John Pearson, Galen Poss
Group President/Specialized Technologies Regina Ridley
Sr. Vice President/Creative Technologies KoAnn Vikören
Sr. Vice President/CIO Lynn Reedy
Sr. Vice President/Human Resources Macy Fecto

BPA INTERNATIONAL

BIT Blasts

News from the World of Game Development

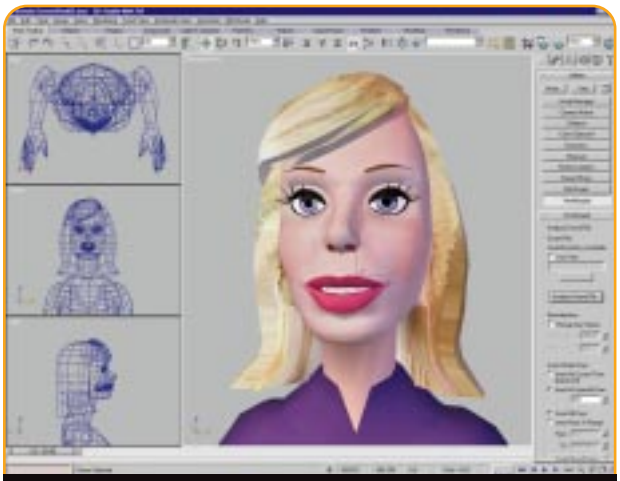
New Products

by Jennifer Olsen

But Does It Walk the Walk?

LIPS INC. has developed Ventriloquist, a plug-in designed to aid animators by automating the complex task of lip-synching facial animations to audio files. Typically, facial animation is a grueling process for animators. To do it correctly, one practically needs an advanced degree in linguistics to figure out all the phonemes, visemes, and every other -eme one needs to master before a speaking character is ready for its close-up. What Ventriloquist aspires to do is to let animators plug character dialog straight in from one of numerous standard audio file formats, and then quickly and painlessly deliver an output stream of precise morph targets.

The immediate results from this process look admittedly stilted, but a one-size-fits-all answer to the infinite range of human (or animal, or robot, or alien) emotion and expression is a tall order. By the time animators are ready to keyframe the finishing touches they've already saved a lot of time, which they can then spend tweaking those faces to perfection instead of sweating the set-up.



Ventriloquist brings easy lip-synching to Max.

Ventriloquist is appearing first for 3D Studio Max 3 at a suggested price of $595. Versions for Lightwave, Softimage, Maya, and others are to follow soon.

■ **Lips Inc.**
Cary, N.C.
(919) 468-7005
http://www.lipsinc.com

Common Environment Still Uncommon for Consoles

MICROSOFT has released the Windows CE Toolkit 2 for the Sega Dreamcast. With the advent of the Dreamcast, console development is finally emerging from its smoke-filled room, no longer the arcane process associated with the current, aging generation of consoles. Sega is banking that it will be able to attract more developers already familiar with the Windows CE development environment to their new console.

Windows CE for Dreamcast means that porting games between Dreamcast and PC will be relatively easy, and is sure to encourage more simultaneous cross-platform development since many of the hardware barriers of traditional console development will be mitigated or eliminated altogether. Memory-light but feature-rich, the Toolkit of course incorporates the DirectX library of APIs and is compatible with Visual C++ and Visual Studio.

One new feature of the Toolkit that may prove interesting for Dreamcast development is its browser support. Internet Explorer 4 HTML Control enables developers to give players access to HTML content from within a game so they can post high scores, link to hints and cheats, and lose some of that excess disposable income with e-commerce functionality.

■ **Microsoft Corp.**
Redmond, Wash.
(425) 882-8080
http://msdn.microsoft.com/cetools/platform/support.asp

Another Loopy Trip for Musicians

SONIC FOUNDRY introduced a new version of Acid Pro, its groundbreaking loop-based music production tool. When it debuted in 1998, Acid thrilled music professionals with its ability to change tempos without altering pitch.

New to Acid 2 is Sonic Foundry's XFX 1 DirectX audio plug-in which enables real-time tweaking of mixes. It also includes Sound Forge XP 4.5, a digital audio editor that lets users create and edit loops, synchronize audio and video, and manage file conversion. Hundreds of royalty-free loops are available that users can simply drag and drop from the explorer window into their track view and arrange into multiple-track creations. Acid automatically adjusts the key and tempo of incoming loops to keep things from getting out of whack, which means less dirty work for you, the artiste.

Once your magnum opus is complete, you can output your music to .WAV, .MP3, .WMA, or .RM files, export as digital audio tracks, or burn it on CD with track-at-once CD burning. Acid Pro 2 is available for Windows 95/98/NT 4.0 and carries a suggested price of $399.

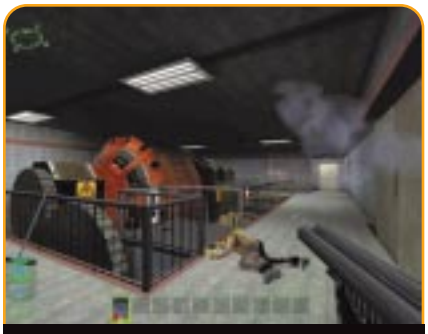
■ **Sonic Foundry Inc.**
Madison, Wis.
(608) 256-3133
http://www.sonicfoundry.com

Industry Watch

by Daniel Huebner

PLAYSTATION GOES DESKTOP. According to some reports, Sony plans to follow the March release of the Playstation 2 with a series of desktop workstations based on the console's Emotion Engine processor. SCEA chairman and CEO Ken Kutaragi noted that the current Emotion Engine is equal to Intel's Pentium III in transistor count, and he expects the next-generation Emotion Engine 2 to surpass the Pentium when it's released in 2002. The workstation will be aimed primarily at users in broadcasting, film production, and software development. As we went to press, Sony had yet to comment on an OS, but it is suspected that Linux will power the new systems.

CODEMASTERS PICKS UP YOSEMITE. A closed studio will gain new life in the so-called "birthplace of computer gaming." U.K.-based Codemasters has announced a plan to open a studio in Oakhurst, Calif., a town that gained notoriety as the home of Sierra's first headquarters 20 years ago. Much of the staff and management of the new studio will be veterans of Sierra's Yosemite Entertainment studio. Craig Alexander, who served as Yosemite's general manager for nearly five years and directed games such as PHANTASMAGORIA and POLICE QUEST: SWAT, will lead the group. The studio is expected to start with two dozen employees and ramp up to 70 in the coming months. The group will keep the Yosemite name, which was purchased from Havas, and will pursue development of both PC and PS2 projects.



AMEN: THE AWAKENING has gone into hibernation, at least for now.

TITUS TAKES VIRGIN INTERACTIVE. Acquisitive French publisher Titus continued its expansion by purchasing a controlling interest in Virgin Interactive, a move that should further bolster Titus's position in Europe. The firm gained a 43.9 percent share in Virgin as part of an earlier deal to take controlling interest in Interplay. Titus purchased additional shares to expand that stake to 50.1 percent and a controlling interest in the company. The terms of the sale were not disclosed, and neither company has yet commented on its plans.

SIERRA PULLS PLUG ON BABYLON 5. In what was billed as a move to "enhance focus on market success," Sierra killed off several titles and eliminated 105 jobs. The cancellation of DESERT FIGHTERS and PRO PILOT PARADISE at Dynamix in Eugene, Ore., sent 60 employees packing. An additional 45 jobs were lost with the cancellation of BABYLON 5, a title that had been relocated to Bellevue, Wash., after Sierra shuttered Yosemite Entertainment earlier this year. Other titles cut in the restructuring included ORCS: REVENGE OF THE ANCIENTS and the persistent-world project MIDDLE-EARTH. The reorganization will ultimately see Sierra divided into three business units: Core Games will focus on Sierra's high-profile games and includes HALF-LIFE publisher Sierra Studios, Impressions Games, Papyrus, Sierra Northwest Studios, and the remaining teams at Dynamix; Casual Games will focus on Hoyle card games as well as hunting, fishing, and rodeo titles; and Home/Productivity will deal with cooking, gardening, and genealogy titles.

AMEN DESIGN STAFF DUMPED. Cavedog Entertainment, well-known maker of strategy games TOTAL ANNIHILATION and TA: KINGDOMS, cited problems with the technological development of its much-anticipated shooter AMEN: THE AWAKENING as the basis for its decision to let go of the entire AMEN design staff and push back the game's release date. While Cavedog hasn't elaborated on the game's problems, the design staff were praised for their "talent, dedication, and contribution" and were invited to return to the project when, and if, the development difficulties are resolved.

BALLARD RESIGNS FROM 3DFX. L. Gregory Ballard tendered his resignation as chief executive officer of 3dfx effective October 31, 1999. Ballard spent three years at the company, seeing revenues grow to more than $400 million. "I truly believe that the challenges in this next phase of the company's growth will be more technical than marketing and strategic, and that 3dfx can benefit from the fresh perspective that a new CEO can bring," said Ballard. 3dfx is forming a search committee, which Ballard himself will lead, to find a replacement. ■

UPCOMING EVENTS CALENDAR

Digital Content Creation

LOS ANGELES CONVENTION CENTER
Los Angeles, Calif.
December 6–8, 1999
Cost: variable
http://www.dccexpo.com

Game Developers Conference HardCore Technical Seminars

HYATT REGENCY
SAN FRANCISCO AIRPORT
Burlingame, Calif.
PHYSICS: December 6–7, 1999
GRAPHICS: December 8–9, 1999
Cost: $1,950/each; $3,300/both
http://hardcore.gdconf.com

Game Developers Conference 1999 RoadTrips

WYNDHAM GARDEN HOTEL
San Rafael, Calif.
December 10, 1999

MEYDENBAUER CENTER
Bellevue, Wash.
December 14, 1999

Cost: $120 ea. (discounts available)
http://roadtrips.gdconf.com

Nichimen's Mirai

by Jeffrey Abouaf

If you attended Siggraph '98 in Orlando and rested your feet at the Nichimen booth, you saw the promise of something new in character animation: Mirai. The promise was realized with the product's release last May, followed by unanimously positive critical reception. It brings many firsts to character animation, and is well worth looking at whether your passions run to cinematic or real-time 3D.

The most noticeable innovation is the degree of skeletal intelligence built into the animation module. The first demo involved a generic bipedal character performing a gymnastic exercise — taking two running steps toward a wall, kicking one foot against the wall to push up and off into a backward somersault, catching a trapeze, then dropping to the floor. The naturalistic motion sequence took about two minutes, required only eight keyframes, and was actually usable. The second part of the demonstration involved the artist refining and smoothing the model, texture-mapping it, and adding touch-up paint, all from within Mirai, with changes updated across all modules in real time. No doubt that first demo belied the technological advances under the hood, because Mirai didn't ship until the following May. Mirai's power and advantages are more subtle and far-reaching than the demo showed: it begins with a working environment, which strives to be more like a 3D operating system than a user interface, and builds on this with a comprehensive, advanced feature set.

A 3D OPERATING ENVIRONMENT. Some leading 3D applications have a UI organized into modules (modeling, animating, rendering, and so on); others perform all operations within a single perspective window, enhancing this with a series of modeless dialog boxes. Each approach has its strengths and limits. Of the former, few, if any, let you work in more than one module at a time; of the latter, the single interface is usually supplemented by floating dialogs. Mirai's designers conceived the interface as a 3D operating system, in which modules behave like applications, each complemented with its own floating dialogs, yet all are dynamically linked so changes to one propagate through the others instantaneously.

This means, for example, you can have multiple geometry editors, 2D paint sessions, 3D paint sessions, and UV mapping windows open simultaneously. In each geometry window, however, you control what is visible vs. what is hidden — that is, you can display different objects or sets of scene objects in different windows, even though you're looking through the same camera from the same vantage point. Having isolated objects and groups this way, you could bring up one or more 2D and 3D paint windows showing the isolated objects. Because these editors are linked, as you paint in 2D it updates both the 3D view and any geometry editors.

This differs from working in an application such as 3D Studio Max, in which you can have more than one instance of a single viewport, but could not hide or show different objects in each one. (In Max, you'd achieve isolation with additional cameras in the scene; in Maya, you'd accomplish a similar result by assigning objects to different layers.) Mirai operates from a single-camera perspective; you look through this camera at all times. (Additional cameras are planned for a future release.) While you will occasionally set the camera to the XYZ orthographic views, the traditional four-windowed orthographic presentation is not the intent. To model and animate in Mirai, you see the scene through the camera lens's perspective. You have a wide choice of lenses, and can save multiple viewpoints, animate camera motions, and attach the camera to a path.

Mirai is "selection driven" as opposed to "tool driven." All 3D programs have you select objects, faces, edges, or vertices, and then perform an operation on them, but generally you choose a tool first. In Mirai, your selection defines your options to a greater degree than in other programs — if you click on the geometry view while working on the model, you can switch to camera mode where all actions change your viewpoint, then click back on an edge or polygon to bring up a menu of everything you can do to that face. When you first encounter this, you may find it so seamless that you inadvertently switch from camera mode to object editing without warning — quite disconcerting. Also, when you're used to manipulators constraining transform operations, it's a little awkward trying to control manipulations in a perspective window. Once you understand the orders, however, this is a remarkably fast way to work.

Although available for both IRIX and NT, Mirai reflects its IRIX roots. Nichimen's N-World (Mirai's predecessor) was deservedly recognized as a premier real-time 3D character animation system. This was when most real-time developers used the SGI platform exclusively. When they ported N-World to Windows NT three years ago, its hefty price tag and high-end hardware requirements kept it out of the hands of all but the most dedicated, well-funded game shops.

Mirai's "SGI" style comes through in its clean interface, with minimal icons and no tool tips; you drive it with left, middle, and right mouse clicks, with results dependent entirely on sequence and context. It supports hot keys, but has far too many commands to be hot-key driven. This is wonderful for the initiated, but will cause consternation for newcomers. The good news is that Nichimen has anticipated the

Jeffrey Abouaf is an artist, animator, and instructor who appears at home in a couple of institutions in downtown San Francisco.

problem with superior printed and online documentation and tutorials. You'll save a lot of time and minimize frustration if you follow the "Getting Started" manual before exploring on your own. By the time you're through with it, you're comfortably reoriented and ready to play.

ONE HELLUVA MODELER. Because of its roots in real-time 3D, Mirai has one of the most extensive modeling toolsets available. The ability to align and bevel vertices, edges, and faces enables precise, easy control of any object property. In the hands of an experienced artist, edges are quickly aligned to follow a character's head and body contours and muscle formations ("Edge Loops"); when these edges are extruded and the mesh area subdivided and smoothed, you achieve high detail and control that rivals any package. Figure 1 demonstrates the smooth yet highly detailed work achievable in Mirai.

The 2D and 3D paint modules complement the extensive materials editor and mapping capabilities. Like many competing packages, Mirai supports UVW mapping parametrically by projection and face-mapping. You can assign multiple types of mapping coordinates (such as planar and spherical) to the same selection set, and you can composite multiple layers of materials on a face selection, taking advantage of different types of projections. In addition, Mirai's painting capability allows you to paint over any seams that might result where different mapping coordinates meet.

Nichimen's August 1999 update to Mirai introduced "magnet moves" along face normals with falloff. This feature allows artists to model by painting surface deformations and displacements. This is handy, for example, for painting extrusions on edge loops to create cheekbones, brows, and so on, or for painting an extruded layer of clothing or armor on a character.

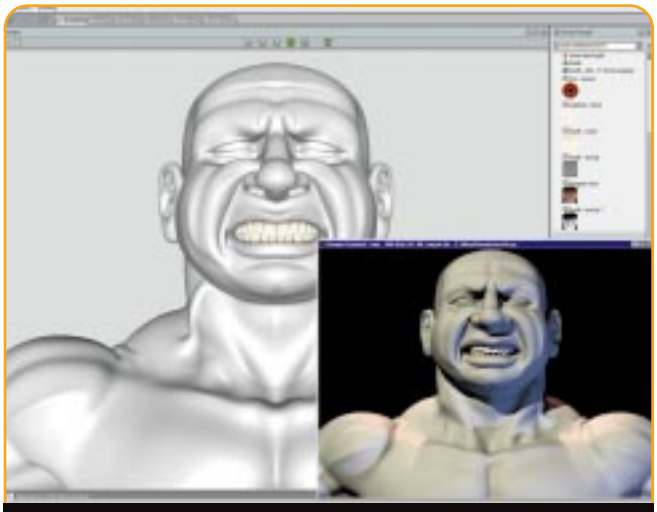


FIGURE 1. *This character shows off the smooth and highly detailed work achievable in Mirai.*

SUBDIVISION MESH MODELING — NO NURBS, B-SPLINES, OR H-SPLINES ALLOWED. Because if its real-time 3D roots, Nichimen has always been a polygonal modeler and never supported spline technologies. Building on the pioneering efforts of Symbolics, and boasting a staff whose credits go back to 1980's *Tron*, Mirai's developers were among the first to embrace subdivision-mesh modeling as their technology for delivering smooth, organic surfaces, only they followed this approach before it became fashionable. They call it "Volume Modeling," and the surface the "Derived Surface"; it amounts to making a low-resolution geometric form, and a reference copy with high-resolution smoothing applied. Changes to the control volume update on the high-resolution version. The value, of course, is that working with polygons presumes you can generate multiple levels of detail (LODs) from the same control volume, that surface deformations do not change the face count, and that polygons provide the lowest overhead for texture mapping. Yet the control volume also acts like a lattice deformer, in that generating poses or morph targets is a very elastic experience.

FK/IK ANIMATION. Mirai's FK/IK solution is to create a stop-motion puppet. Rather than keyframing each skeletal joint rotation, keyframes are set for the entirety, pose to pose. The IK system uses quaternion algorithms similar to those used in other packages, with a second-pass analysis checking for incorrect or less-than-natural motion. Whereas other applications have skeletal motions dependent upon pulling the skeleton along a path, Mirai is about altering poses. The poses can be set by rotating joints and/or by "pinning" bones. This pinning can be in the nature of "glue" or a "tack"; if you glue a left foot and pull a right hand, the foot will remain in place as long as possible before pulling away. If you tack a foot, the foot will not move and the hand will move until it slips away. The pins can be temporary; different bones can be pinned or unpinned at different frames.

Second, Mirai's skeleton responds to "magnet" moves, that is, you can

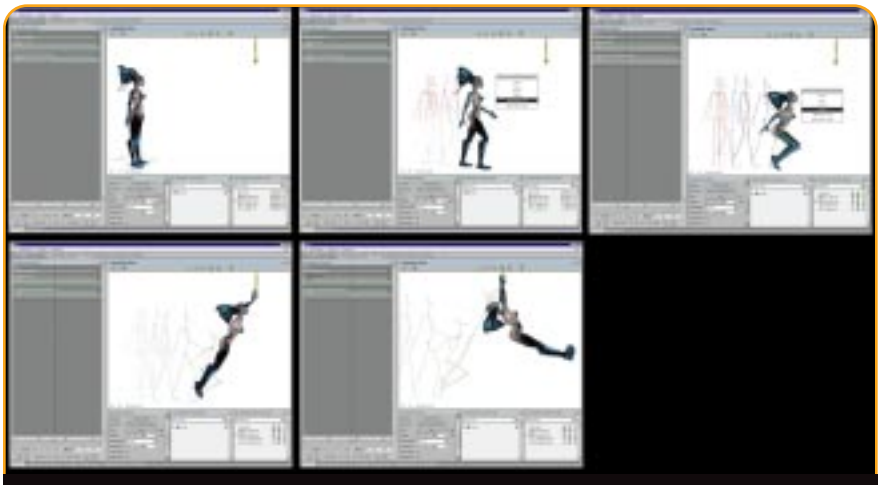


FIGURE 2. *Combining Mirai's various animation effects offers naturalistic results for complex animation sequences.*

move one bone relative to another specified bone. This allows naturalistic squash and stretch. These features combine, for example, to let the character crouch (magnet move), spring upward (pinned feet); loop around the trapeze (feet unpinned, hands pinned), and drop into a crouch. The fairy sequence in Figure 2 shows how these work together.

Mirai supports the most common motion capture formats (such as Acclaim, Motion Analysis, and Biovision), and can send the data out to Nichimen's Game Exchange utility. Nichimen's current development direction is expanding this biomechanical capability. In addition, Testarossa (which won an Emmy award for its figure skating simulations for the most recent Winter Olympics) has written a set of mo-cap plug-in tools for Mirai, designed to extract added functionality from the data files (for more information, see http://www.toolsinmotion.com).

Finally, Mirai supports full nonlinear editing of motion and motion capture data. For example, you can set up a run cycle and loop it (using motion capture data or keyframing), then add a second layer of dodging and weaving obstacles. The two motions combine seamlessly into the final sequence, but could also be taken apart and recycled. Mirai also lets you stitch two motion sequences together, and gives you full control over the transition. Figure 3 shows the animation editor set up for a transition between two motion clips. Not only can you control the speed, number of frames, and other characteristics for the transition, you can layer other motions on top of the source clips. Further, with Mirai's scripting capabilities you can call and recombine additional premade layers or scripts to make complex, unique results.

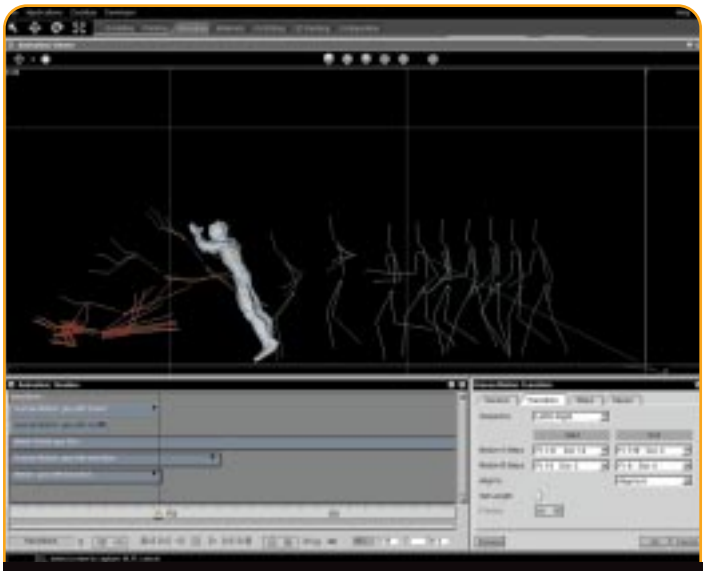


FIGURE 3. *Mirai allows you to stitch two motion sequences together and retain control of the transition.*

FACIAL ANIMATION: DISPLACEMENT EQUALS LIP-SYNCH.

"Displacement" in Mirai is what other packages refer to as morphing or blend shapes — in each case it's isolating a series of facial expressions for direct or indirect use as morph targets in facial animation. Mirai has this capability, as did N-World before it. The documentation shows how to set up the displacements, how to "wire" them to sliders, and how to use the sliders to generate unique expressions. Because motion editing supports layers and scripts, setting up displacements for one character implies you can adapt these to a different characters without starting from scratch. Figure 4 shows Bay Raitt's Horus character set up for lip-synching. Note you can see the smoothed version, the connected low-polygon modeling version, and the slider-driven animated composite all within the working environment, together with the graphs and timeline pertinent to Horus's speech.

Figure 5 shows John Feather's Grunty character, specifically how you would set up to animate him in Mirai. The facial features have been wired to sliders in the Animation Mixer, the character's hierarchy is laid out in the Animation: Viewer Graph, and we see the

FIGURE 4. *A character set up for lip-synching. All the information pertaining to the character's speech is shown.*

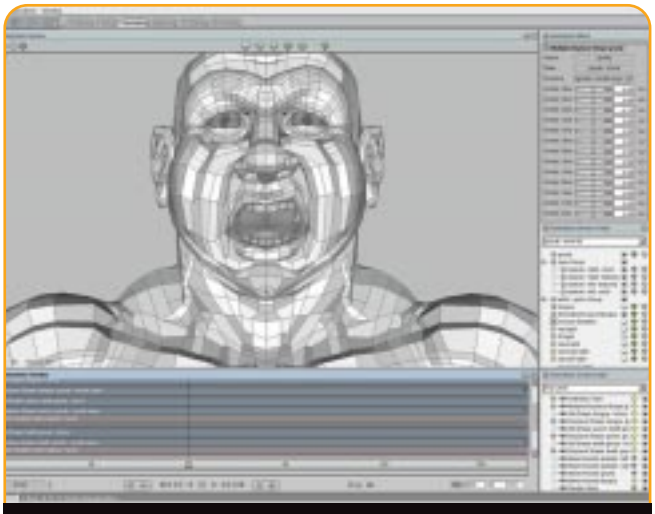


FIGURE 5. *A typical facial animation setup in Mirai, with all the relevant information at the user's fingertips.*

FIGURE 6. *The facial animation results: a high-resolution version of the character with four morph targets.*

hierarchy of movements in the Animation: Script Graph and the Animation Timeline. The results appear in Figure 6 which shows the smoothed, high-resolution model of Grunty, together with four morph targets.

PARTICLES, RIGID DYNAMICS, GELATIN, AND ROPE. Mirai supports both rigid and soft-body dynamics, allowing you to set up interactions between objects such as collision detection, wind, gravity, and so on. The soft-body dynamics include Gelatin and Rope simulations. Gelatin is intended to simulate jiggling, as in the belly of a fat man or something more exotic/erotic by the heroine/hero. Rope calculates the effect of a tire tethered to a tree branch, and should be very useful where any two objects or characters are bound together. The particle system responds not only to forces and turbulence, but can work into simulations as any other object. Further, particles can be the built-in standard shapes or any geometric shape.

OUTPUT: COMPOSITING AND LAYERING ANIMATION. If Mirai has a weakness, it's the renderer. To be fair, however, this weakness in Mirai simply means that the built-in renderer does not rival Renderman or Mental Ray 2. This review did not pit Mirai's renderer against the new renderers built into 3D Studio Max 3 and Maya 2, but it certainly measures up to the renderer included in the first releases of these competitors, and exceeds the quality of many other packages. Like the best of the renderers, Mirai's is multi-threaded. For the professionals, Mirai will be adapted to work with Renderman on IRIX and NT by year's end.

You can render to stills, play them back in real time, and/or compile them into a movie. Mirai includes compositing tools for layering animation and combining work with live-action output, titling, and the like. While the post-production tools are not nearly as extensive as the modeling tools, and lack the state-of-the art reflected in its FK/IK, Mirai's output tools reflect the fact that this product has earned a place in both cinematic and real-time 3D production.

CONCLUSION. Nichimen held its first annual Mirai user meeting at Siggraph '99. While I was not surprised to see several hundred show up, I was surprised at how many were serious game artists (averaging ten years' experience), many of whom have been devotees of the competing high-end 3D modeling/animation packages, who said they had switched to Mirai for character animation production, or would be soon. The consensus among the faithful was that Mirai's FK/IK capabilities are at least one generation ahead of the competition. When I brought this up with the many representatives of competitors at the show, they made it clear they're watching Mirai.

Mirai is a solid, versatile package with cutting-edge character animation capabilities. With its highly evolved FK/IK, biomechanics, and facial animation capabilities, it's ready for mainstream use in games and all real-time 3D. But to limit it to this niche is to underplay its full capability, because the modeling, texturing, painting, and dynamics properties make it as viable for prerendered use, and compatibility with Pixar's Renderman targeted for year's end can only enhance that. The obstacles facing Mirai's entry into the production pipeline appear to have little to do with the product, because it looks able to compete with the best. Building a large user base and replacing existing production systems will prove the biggest barriers. ■

Acknowledgements

Special thanks to Intergraph Computer Systems for use of a TDZ workstation, and to 3Dlabs for their GVX1 card used to test Mirai. No virtual humans were damaged in the course of preparing this article.

Nichimen's Mirai: ★★★★½

Nichimen Graphics
Los Angeles, Calif.
310-577-0500
http://www.nichimen.com

Price: $6,495. Nichimen also offers a 90 percent educational discount.

System Requirements: Windows NT 4.0 or SGI IRIX 6.3. Both operating systems require 128MB RAM, 300MB disk space, 300MB virtual memory, three-button mouse. Windows requires 266MHz Pentium II plus OpenGL accelerator.

Pros:

1. "3D operating system" interface enables multiple windows of the same type, revealing/hiding different objects, and hot-linking to other modules such as paint.
2. Intelligent IK/FK system is very easy to use in final animation.
3. Motion capture and motion layering; sophisticated support for main motion capture systems.

Cons:

1. NT users may have difficulty with IRIX-inspired interface.
2. Rendering engine as it is now is adequate, but not world-class.
3. Needs larger user base before cross-user support and third-party plug-in development get fully underway.

A Clean Start: Washing Away the Millennium

The millennium is coming to a close at the end of the month. While I actually believe that the turn of the century is a year from now, I am going to fight it no longer. The Y2K hype has washed away any chance of restraining the feeling that something big is about to happen.

I for my part am going to toast the new millennium *next* year, but who wants to miss out on the big party? Let's make it a yearlong celebration. What the hell, it only happens once every thousand years — I'm going to enjoy it.

For the doomsdayers, it's going to be a year of hunkering down in their battery-powered shelters, hoarding food, waiting for everything technological to spiral out of control. Many will be nostalgic for the grand old days of mechanical cash registers and supermarket checkers who knew the actual prices. However, I look forward to the new year. I like all things techie and don't really care if they get the date wrong. To quote Douglas Adams, I still think digital watches are a pretty neat idea. The year 2000 is going to be full of exciting new toys for game developers to play with.

In fact, if you watch the news you'll know that game developers are going to get equipment so sophisticated that the government considers them weapons. This will be the first year that we create games for home machines that can perform more than two billion operations per second. Some of the press I've read said that developers aren't ready to handle that much power. I don't know what sources they have been talking to. I have never met any developers who didn't think they could use more power even on their current projects.

Hype and Demos

That brings me to something that I find annoying in the industry. News in the high-tech business seems to surf hype waves constantly. Nothing gets any press unless it has "never before been seen on a game console!" or is "unlike anything ever created in a computer game!" These quotes generally come from very early views of technology demos or art tests. Any evidence of game play or interactive experience is completely missing.

Game companies realize this so a great deal of effort is spent just coming up with the technology. A flashy demo can get them press coverage, attention from publishers, and that intangible "hype" from the gaming public. Some of this forms the backbone of a healthy development cycle. However, when the game engine is directing the development of the game, priorities are out of whack. Most gamers fondly recall games that lacked flashy technology yet captured their souls for hours.

When it became public that QUAKE 3: ARENA would support a form of curved surface geometry, suddenly this became the must-have feature for 3D action games. Games were considered to be using "old" technology if their engine didn't support curved surfaces. This happened regardless of whether the game had environments that would benefit from curves. Programmers had to go back and retrofit their engines with some kind of parametric surfaces. Level designers had to go back and invent places that would benefit from some kind of curves. Didn't matter what they were or where they were used, just as long as it could be added to the feature list.

It should be clear to anyone who has read this column before that I believe technology can be a strong force in creating a more compelling game experi-

Whether he's testing the theories of water displacement while scuba diving or in the tub with his duckie, you can drop Jeff a line at jeffl@darwin3d.com.

ence. However, technology should not be turned into a checklist and become the only determination for a game's potential for success. The game is what is important. The technology is just a vehicle to enhance the experience for the player.

At E3 this year, I was amazed at the Playstation 2's reception. There we were at a show filled with amazing games for all platforms. Real-time 3D graphics were represented everywhere. The level of art was unbelievable. Game play and production on every platform from PC to Game Boy were very impressive. But what was the "buzz" in the press? The Playstation 2 demos. The Playstation 2 is going to revolutionize gaming. Games will never be the same. What interactive demo was shown that elicited these opinions? Some footage of a car driving through a scene and a duck in a tub of water.

Sure, it was a beautiful car and a cool duck. However, it was just a display of technology and computing power.

Don't get me wrong. I am certain the Playstation 2 will be an amazing console. The hardware looks impressive. Sony has definitely proved they can foster the creation of great games and lots of them. They also wanted to build up momentum for the new console, and I think they succeeded. They recognized that technology can be used to build hype and anticipation without even having a game.

LISTING 1. *Processing the height field.*

```
////////////////////////////////////////////////////////////////////////////////
// Procedure:       ProcessWater
// Purpose:                 Calculate new values for the water height field
////////////////////////////////////////////////////////////////////////////////
void CAguaDlg::ProcessWater()
{
//// Local Variables ////////////////////////////////////////////////////////
        int i,j;
        float value;
////////////////////////////////////////////////////////////////////////////////
        for (j = 2; j < WATER_SIZE - 2; j++)
        {
                for (i = 2; i < WATER_SIZE - 2; i++)
                {
                        // Sample a "circle" around the center point
                        value = (float)(
                                READBUFFER(m_ReadBuffer,i-2,j) +
                                READBUFFER(m_ReadBuffer,i+2,j) +
                                READBUFFER(m_ReadBuffer,i,j-2) +
                                READBUFFER(m_ReadBuffer,i,j+2) +
                                READBUFFER(m_ReadBuffer,i-1,j) +
                                READBUFFER(m_ReadBuffer,i+1,j) +
                                READBUFFER(m_ReadBuffer,i,j-1) +
                                READBUFFER(m_ReadBuffer,i,j+1) +
                                READBUFFER(m_ReadBuffer,i-1,j-1) +
                                READBUFFER(m_ReadBuffer,i+1,j-1) +
                                READBUFFER(m_ReadBuffer,i-1,j+1) +
                                READBUFFER(m_ReadBuffer,i+1,j+1));
                        value /= 6.0f;                  // Average * 2
                        value -= (float)READBUFFER(m_WriteBuffer,i,j);
                        // Values for damping from 0.04 - 0.0001 look pretty good
                        value -= (value * m_DampingFactor);
                        SETBUFFER(m_WriteBuffer,i,j,(int)value);
                }
        }
        SetDisplay();                   // Draw and Swap Buffers
}
```

Yes, I Have a Point

I want people to realize that technology is just a means for achieving a result. I hope to discuss techniques that make you think about ways you can attack a particular problem. Hopefully, these ideas can be used on a variety of platforms in a variety of ways.

For example, let's take the water effect from the Playstation 2. I think one of the reasons it was so impressive is that water is terribly difficult to represent in a real-time 3D simulation. It remains hard even for the visual effects community. They have unlimited polygon budgets yet modelers will still groan if you ask for realistic water. In the early days of real-time 3D games (what was that, like three years ago?), polygons were at a premium. You could not represent water with lots of geometry so developers had to create animated textures that showed nice rippling water. It was still pretty lifeless. It could not react to what was going on around it and sort of just did its own thing.

However, with all the billions of operations per second the new millennium will offer to developers, we can do better. You probably have seen waterlike ripples as Java web site banners. The effect has been around for years and is quite simple. It does a good job of simulating how ripples will interact with each other and reflect off barriers. Since it can be created in Java, it's obviously not too complex, either.

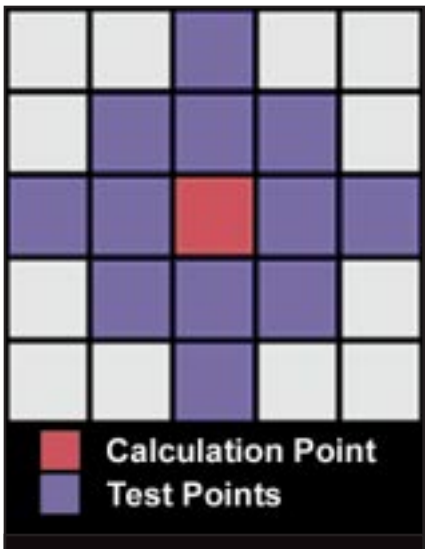


FIGURE 1. *Sampling the water field.*

In fact, the effect is really just a form of image processing. Start by creating a double-buffered height array that will hold the values for the water level at each position in the grid. The key to making this array behave like water is to determine the new level at each location. This is done by sampling the surrounding locations to determine whether the current location should be moving up or down. I chose to sample a rough circle around the center point as you can see in Figure 1.

If I wanted to average the water levels over this region, I would add the values together and divide by 12. However, this is where we are going to fake some of the fluid dynamics that make this look like water. Water level actually rises when the surrounding pressure is increased. Think of squeezing water in a plastic bottle and watching the water in the center rise. So, I can think of the water level at each location as representing the water pressure. When the water level surrounding a location is high, that has the effect of raising the water level at the center above the surrounding locations. Likewise, when the surrounding level is low, the pressure is greatly reduced and the level at the center should drop below the average. So, instead of dividing by 12, I divide the sum of the surrounding levels by six, doubling the average height.

In order to give the water motion, the height of the current position in the previous frame is subtracted from the new calculated height. Now everything is in motion. However, there is no way to reduce that motion. I can cause the system to lose some energy by applying a damping factor that is multiplied by the change in height. That way, I can be sure that the field will come to rest if nothing is changed manually. The code for calculating the water level is in Listing 1. To get things started, I simply write some values directly into the height array and let it run.

Viewing the Water

I now have a height array that animates in a way that forms ripples and wakes. I can easily turn that height field into an image and display it by converting the values to grayscale or to some color scheme. This image ends up looking like a bump map. You can see what it looks like to convert the height array into an image in Figure 2.

This creates a pretty good texture that could be used in a 3D environment to simulate a pool or fountain. However, I can make it even more interesting by applying some environment mapping techniques. The gradient of the water levels surrounding a location can be used to define "normals" like you would find on a 3D mesh. I could then trace these normals to see where they intersect my environment map. However, an even faster way is to treat these gradients as offsets into the environment map. At position (u,v) in the Height array:

```
offsetX = height_array(u + 1, v) - height_array(u - 1, v);
offsetY = height_array(u, v + 1) - height_array(u, v - 1);
sourceTexel = (u + offsetX, v + offsetY);
```

This can be blended with the height color to create a shaded reflection, as you can see in Figure 3. However, this kind of blending is processor-intensive. But, since I eventually want to use this in a real-time 3D environment, why not make use of my graphics card?

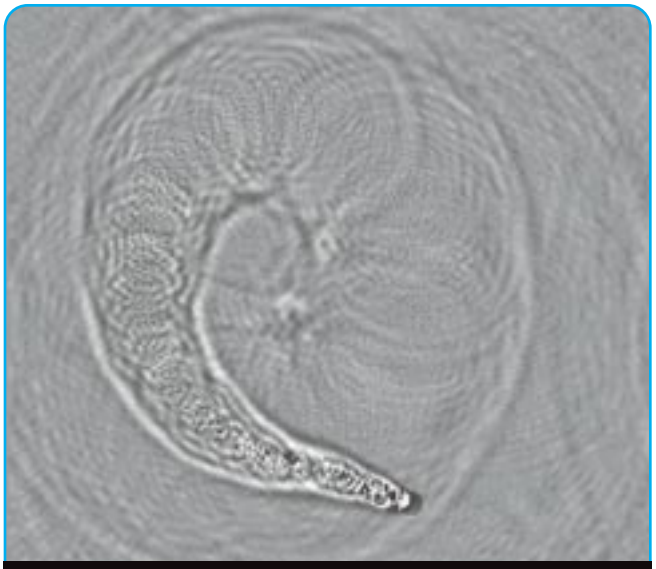


FIGURE 2. *A water image.*

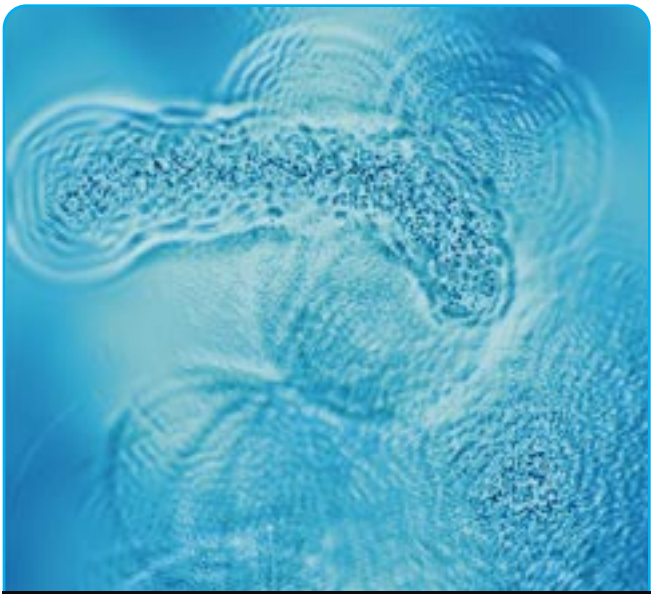


FIGURE 3. *Environment mapping and shading.*

Hardware Help

I have a nice 3D graphics card that can blend two textures together without involving the CPU. To make this work with my textures, the environment map calculations set the UV coordinates for the environment map pass. Most of the graphics hardware that is currently popular with game players can blend two textures together in a single pass. This means that if your hardware can handle it, the blending of the environment map and bump map are rendered at once.

Once hardware is in charge of the blend, it's easy to control the amount each image contributes to the final render by using the alpha values. I can also take advantage of the built-in filtering to smooth out the fact that my maps are of relatively low resolution (say, 128×128).

I can also use the information I now have to make the scene even more immersive. Water consisting of a single

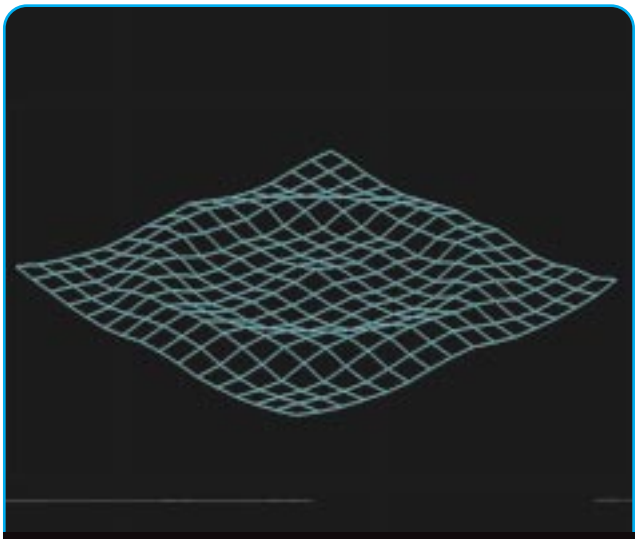


FIGURE 4. *Water made from springs.*

flat polygon looks strange no matter how interesting the animated texture looks. However, I could create the surface on the same grid as the height array. If my height array is 128×128, I can create a polygonal grid that is also 128×128. I then use the data in the height array to displace the vertices in the water mesh. Now, when the water ripples, the texture not only changes, but the actual surface of the water moves also. This all combines to create a very realistic looking water simulation.

Best of all, there are built-in performance adjustments to make sure it runs well on all sorts of systems. If the player doesn't have fast 3D hardware you can use a lower resolution for the displacement mesh. If the CPU is older, lower the resolution in the original height array. You can also turn off the environment map if fill rate is an issue. If the player has a really nice system, all features are set at the highest resolution. This kind of detail adjustment is critical as the number of possible user configurations increases.

Improving the Accuracy

If I generated actual normals, I could improve the look a bit. For one thing, I could simulate the light diffraction through water. I could then create another environment map representing the reflection, which is underneath the water. I'm not sure that it would look any better than the environment map hack I already described. However, actually using the normals to generate the map for the reflection makes some sense. Real environment maps are view-dependent and require the normals for calculation. The next generation of graphics cards coming out this year support cubic environment mapping. That will enable effects such as water reflection to be calculated more easily.

I saw another effect recently that really adds to the realism of rendering water. Nvidia has been showing a demonstration of Fresnel Reflections where they create a completely variable reflection based on the viewing angle. It produces a much more accurate reflection image at the cost of extra CPU calculations. They may have some information on their web site about the technique by now.

Considering the actual physics of the water itself, the method I described here is completely inaccurate. There is no physical simulation of water happening at all. Like many game programming methods, tricks are used to achieve the desired look without the complex calculations.

These days, however, it's worth considering designating more CPU power for methods that increase the dynamic realism in the scene. When I wrote the column on mass-and-spring systems for cloth animation ("Devil in the Blue Faceted Dress: Real-Time Cloth Animation," May 1999), it may have occurred to you that a spring system could be made to behave like water. Each element in the mesh could be restricted to move only up and down and each element is connected to its neighbors by springs. You can see this in Figure 4.

However, even this method differs from the way actual water performs in several ways. One key difference is the way true water behaves when compressed. The total volume of the water should be preserved. If I compress the water on one side, the level should respond by rising on the other side. Also, the methods I have discussed do not account for what is in the water. I am interested in simulating the actual flow of the water. It should be able to spill across a terrain as well as form eddies and standing waves. Also, a height field naturally makes effects such as breaking waves or splashes that flow over into adjacent cells impossible. To get these kinds of interactions, I need to make the simulation more complex. I will have to look towards the field of computational fluid dynamics (CFD). Perhaps there are some lessons in that field for game developers.

But that will have to wait until next time. For now, play around with the simple image processing method for water. It is easy enough to understand and you can quickly experiment with a variety of effects. ■

FOR FURTHER INFO

The image-based water effect has been kicking around so long, I don't know where it originated. I see it crop up on Internet forums every couple of months. A quick pass over the search engines yielded quite a few links:

- **http://www.remedy.fi/company/cool_stuff_data/stone1.shtml**
  A Java-based applet implementing something similar to what I described.

- **http://freespace.virgin.net/hugo.elias/graphics/x_water.htm**
  Hugo Elias has a little bit of everything on his site. The section describing the water effect is pretty thorough, complete with a sample application.

Also check out:

- **Fournier, Alain, and William Reeves. "A Simple Model of Ocean Waves." Siggraph 1986, Vol. 20, No. 4, pp. 75–84.**
  I didn't talk about generating wave motion via simple trigonometric functions. This paper examines this in depth.

- **Kass, Michael, and Gavin Miller. "Rapid, Stable, Fluid Dynamics for Computer Graphics," Siggraph 1990, Vol. 24, No. 4, pp. 49–55.**
  If you want to read ahead and learn about real-time methods for CFD, check this out.

No Pain, No Gain: Implementing New Art Technology in QUAKE 3: ARENA

When beginning a new project, an immediate question that comes to developers' minds is, "Are we going to use the same technology or an entirely a new game engine?" A question that pops up for the artist in particular is, "Do I upgrade to the next version of my art tools or stay where I am?" While the answer to the first question often varies, the answer to the second shouldn't.

Spending more than a year or two working on a game means that the tools you started with have invariably evolved into their next version or iteration by the time you have finished. Unless you've switched mid-project, you have to take the time at the end of a project to learn the new tools — that is, if you're given the time to train and the upgrade of the product itself.

Changes from Q2 to Q3A

When we completed QUAKE 2 at the end of 1997, we started in on our next project, QUAKE 3: ARENA. For id, this new project introduced a dramatic departure from the usual first-person shooter formula. Instead of having players opposing a horde of bloodthirsty monsters while struggling to figure out puzzles and find keys that would allow them to escape a labyrinthine maze, QUAKE 3: ARENA would be all about deathmatch. Gone was any sort of suspense or story-driven tension. Instead, the game was about combat and competition — players pitted against other human opponents or complex, artificially intelligent bots via a LAN or the Internet.

During QUAKE 2, I modeled in Kinetix's 3D Studio R4 (DOS) and animated in Alias|Wavefront's Power Animator. The reason I went with Alias is because fellow artist and id co-owner Kevin Cloud had been using it since QUAKE and already integrated it into the production pipeline. Recombined by Carmack's wizardry, the game engine animated characters in the game via vertex deformation using a string of .TRI files exported from Alias as basic, linearly interpolated keyframes. Each animation cycle — walks, runs, deaths, and so on — were stored as separate files that could be accessed and re-exported to .TRI files whenever we wanted. Of course, the immediate problem that comes to mind when creating models and animations this way is that when any changes are made, a plethora of files must be tweaked and re-exported — very painful, trust me.

Since the question of utilizing new technology for a new project is moot at id (Carmack always retools or recreates the game engine for new projects), it came as no surprise when John announced the implementation of a new animation system for Q3A. The new "tag" system would save storage space by using a single triangle to represent certain body parts such as the head and upper body, and would give me more flexibility to create better animations at a higher polygon-per-model budget. This system also enabled more realistic and spontaneous animations, since the lower and upper body animations were completely detached and unrelated unless explicitly specified otherwise.

For me, the new animation system meant I could consolidate the animations for a single character into one file. This in essence rendered the animation file for a character in Q3A a "folder" in which sequences are like pages that the engine references when animating the game's characters. Making changes to the model became a lot easier, to say the least. Another advantage of keeping the animations in one file was that it allowed me to utilize a great feature of Character Studio: sharing .BIP files. Character Studio's ability to stream different animations together via its Flow Manager was one of the primary reasons I chose Max after Q2 instead of Maya, Softimage, Lightwave, Nichimen, or any other program. Sharing animation data is so important because, in a crunch, I can simply plug one character's animation data into

Paul Steed is a Guy. He likes Guy things: working out, playing pool, drinking beer, and trying hard to stay out of trouble. He happens to make art for computer games and occasionally convinces learned editors he's a competent enough writer to contribute to their esteemed publications. Write him at psteed@idsoftware.com if you're bored. Contrary to popular belief, there's no such thing as too much e-mail.

another and make tweaks instead of having to animate 40 characters entirely from scratch. This worked only because the sequence of all the character's animations had to be the same (for example, three death animations, then gestures, then walks, then runs, and so on). The length and style of animations could vary from character to character, but the basic order had to remain the same due to the new animation code.

Match Tag "A" with Tag "B"

In QUAKE 2, each frame of animation I exported served as a keyframe for the engine to interpolate the position of the mesh's vertices linearly to the next saved animation frame. Although the frame rate was only 10 frames per second, some characters had up to 500 frames of animation to support — not only, say, firing a weapon, but firing a weapon while standing and firing a weapon while crouched. There were no animations of characters firing while running or moving, since I couldn't anticipate where in their stride they would be when attacking. Of course, this quickly reveals the limitations of the vertex deformation scheme as the character slides all over the place while firing on the move (hence the term "skating"), because his body is locked in the stationary "firing-while-standing" pose.

Hierarchically, implementing the tag system (see Figure 1) meant that the lower body would be the parent and the upper body the child (since the lower body incorporates locomotion). Getting a character into the game went something like this: create the character, assign the Physique modifier to the mesh, animate it in a specific order of animations (both body parts together, then upper only, then lower), create a small right triangle with the point facing forward, name it TAG_TORSO, link it (again by assigning Physique to it) to the point where the biped spine rotates, and save it. Once I had the file saved I'd delete the upper body and export the lower body only as a .3DS file (the engine couldn't work with the native Max file format), keeping the tag as a representation of the upper body. (It was linked to the skeleton/biped, which the exporter ignored.) I'd then do the same for the upper body.

This sucked — mainly because of having to remember which body part to delete, and the fact that after doing it for a couple of characters the poor artist's mind can get confused. Also, once we got the model to show up in the game world by converting the .3DS files to a game-digestible format, the animations weren't quite right. While it was cool to move the mouse and see the upper torso move with the "free look" motion of the mouse, the upper body was stiff. Once you passed a threshold of 30 degrees or so, the feet would stay still while the body rotated in place. This rotation (like the sliding effect in our previous system) broke the illusion of the character having contact with the ground, exhibiting any sense of weight, and it was generally not a good thing. So we made some changes.

FIGURE 1. *Mesh, skin, and Character Studio biped with proper tag placement.*

Instead of deleting anything, we just kept everything in one file and made sure the different body parts adhered to a strict naming convention. Any file associated with the upper body would be preceded with a "U_" (U_TORSO, U_ARM, and so on). Files associated with the lower body would be preceded with "L_." We also determined that the reason the upper body still appeared stiff was that people naturally lead turns with their heads. So we detached the head, necessitating a third naming convention ("H_"), as well as a new tag (TAG_HEAD). To counter the problem of sliding in place while turning, we added a "shuffle" where the character would pick up his feet and turn programmatically (Figure 2).

FIGURE 2. *To combat characters sliding around in place while turning, a "shuffle" was added to give them something else to do with their feet.*

This was really no big deal as all animations have to be done "in place" for bounding-box consideration. (The bounding volume of a model created by its aggregate vertices equals the area in XYZ space that registers a hit when fired upon.) Another change we decided to make was to export the models as ASCII files instead of .3DS. From the huge amount of information contained in an ASCII file (or in Max's case, .ASE), the programmers could cull all the information they wanted about normals, UVWs, and animation data.

This worked much better. The head detachment as yet another layer in the tag hierarchy made the motion of the character's upper torso correspond almost eerily with the movement of the mouse. As the mouse moved, the head would move just barely, and then the torso, and then the character would raise and lower his feet as he turned in place to look where the mouse was telling him to look. Of course, the drawback to this was that facial expressions and any sort of hair animations went out the window. Still, the in-game animations didn't quite jibe with how I wanted them to look and definitely differed from an orientation standpoint from their Max file counterparts.

Jane! Stop This Crazy Thing!

After some research, it became clear to me why the animations were off — it was the stupid tag. I had been linking the TAG_TORSO to the torso of the skeleton or biped (first spine link). This was wrong. I was supposed to be linking it to the pelvis. (I was confused — sue me.) I also misunderstood the fact that the base of the right triangle of the tag was the exact point at which the rotation needed to occur (not the overall area of the triangle as I had thought).

After I got that straight, things looked better, but if the pelvis rotated at all or was otherwise moving inappropriately, the torso of course did its animations from whatever position the sometimes errant tag told it to. Since the tag represents the upper body's position relative to its predefined default position (a single frame in the animation set), rotating the tag itself during the animations would rotate the base at which the upper body was attached. So, happy with this second level of "after market" animation capability, I went and started rotating the tag so it would always face forward. Soon I became frustrated, however, because the tag would do this weird shearing thing and the triangle would try to skew itself in a seemingly random fashion.

FIGURE 3. *Hours of consternation over inappropriate shearing could have been avoided by turning off the animation tags' Inherit Scale at the start.*

This...drove...me...nuts.

After all kinds of kludgy, inefficient, mickey-mouse attempts at band-aiding the problem, I resorted to bugging Character Studio's developers and asking them for help. After much experimenting and exchanging of e-mails and Max files galore, Jeff Yates at Kinetix asked me if I had unchecked a particular option in the Hierarchy options of the mesh. D'ohh. How the heck was I supposed to know that? (This series of check boxes is in the Inherit Scale submenu of Link Info.) Once I turned the Inherit Scale off for the tag (Figure 3), the shearing went away and I was happy...for a while.

FIGURE 4. *QUAKE 2 shotgun (left) vs. QUAKE 3 shotgun: animated hands and field-of-view-compensated geometry give way to no hands and more attention paid to the weapon's business end.*

Then I noticed some other problems with the torso moving around and, clever guy that I am, I thought about the process some more. You see, the conversion process of .ASE to .MD3 (game format) happened via strict adherence to naming convention. This basically allowed me to have all kinds of props with the file and, of course, made the converter ignore them along with the biped geometry (among other things) through "wrong" naming conventions. Since the tag at the torso was essentially doubled (according to Carmack) as the code separated the upper and lower body, why couldn't I double the pesky tag, name it something besides TAG and use it as reference? This way I could see why the upper body wasn't seating properly on the lower body.

So that's what I did and, lo and behold, there were some discrepancies (mainly due to the mostly goofy IK solution Character Studio uses for the upper body). Utilizing the oft-unsung "snap to vertex" power of Max's Grid and Snap Setting dialog box , I adjusted the position of the tag and made sure it matched my reference tag at the base of their respective right triangles.

This Is My Weapon, This Is My Gun...

Having mastered the basic mechanics of the new tag system it was time (naturally) to make some more changes. People complained about the difficulty of seeing which weapon they held in Q2. Mod authors quickly rectified this by adding customized code that allowed players to see their weapons during deathmatch. In Q3A, we decided to go ahead and implement the multiple weapons, but added a "weapon-switch" animation. This departed from QUAKE's original instant weapon-switch that the diehard fanatics were clamoring for even during Q2 (Figure 4).

Carmack wanted me to use the same animations for all weapons (one default position, one firing, and one weapon-switch). I balked, saying I wanted to have different positions for each weapon so the design wouldn't be dictated by a single grip. Also, how a player held a weapon could be noticed from a distance and be an additional cue for ordinance recognition. Well, I didn't get my way but I did get one extra default position for a melee weapon (the Gauntlet). This sort of compromise occurs all the time when you pit technologically-constraining frugality (keeping the memory requirements low begets a faster-running game, obviously) vs. artistic expression.

Another artistic hurdle was the fact that my programming leader wanted to implement the tag system in the weapon as well. This would save even more space and do what we hadn't done in Q1 or Q2: use the same model for both the "in-use" view of the weapon and the weapon you see in the world. The reason the previous two efforts didn't share the same geometry for both instances of seeing weapons was...well, I don't know.

The only problem with this new method of seeing the weapons was that it added yet another shackle to the weapon design constraints. The field of view while playing the game is at least 90 degrees (it can be raised if desired). While it may not quickly be apparent why this is a problem, the difficulty lies in how much of the weapon you actually see in your view: only the end of the barrel. In QUAKE 2 I used animated hands and squished, unique geometry to make the weapons in view look interesting and, more importantly, to simulate the right perspective.

So, using mostly artist/owner Adrian Carmack's cool designs, I went about making weapon models that were equally appealing in both the normal view and the in-use view. Do I make it centered, held to the right, the left, what? In the end, I settled on a slightly-to-the-right position that corresponded to the animation of the characters when they changed weapons (see Figure 5).

I solved the tag issue by first building a weapon that I positioned in the character's hands with a comfortably aggressive stance, settling on a semi-sniper position. I took this first weapon and embedded a copy of the tag I used for the torso and the head into the weapon. The reason for the tag attachment was so that I could link the weapon to the character's right hand (having positioned it correctly). That way, after I had completed the character's animations and was ready to export the file, I could create a snapshot of the weapon, delete all vertices except for the triangle, rename it TAG_WEAPON, assign Physique to it, link it to the right hand, and finally delete the original weapon (or keep it under a garbage name for reference).

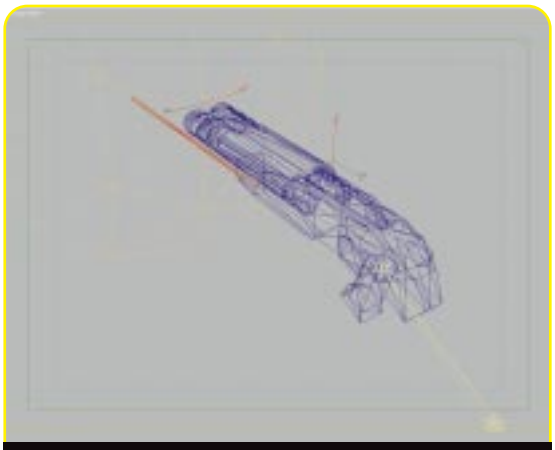


FIGURE 5. *The final design for Q3A's weapons also incorporated the animation tag system.*

As far as the commonly held grip dictating the design of the weapon, I think we ended up with a pretty varied arsenal despite the limita-

tion. Once I completed a weapon I just placed it over the top of my archetypal first weapon (aligning the grips so they matched), deleted all the first weapon's vertices except for the tag again, and saved the new weapon off with a tag in the exact same position as the one in the first weapon. Thus, when I went through the process of animating and exporting weapons, they all had their tags in the right place.

And What About Style?

Enough about the mundane and tedious assimilation/accommodation of the technology into the workflow, what did I actually do with it? Now that's a great question! I made some pretty cool models (textured by the incomparable Kenneth Scott) with some pretty cool animations...and I had to fight for them the entire time. Of course, Carmack's minimalist philosophy towards things such as vertices, faces, texture map size, and total frame counts was at direct odds with my grandiose plans to create some high-flying, John Woo–style, highly kinetic action. Luckily, he's the type of guy who, once shown something, can be convinced to bend a little for art's sake.

A good example of this was my hell-bent desire to have swimming animations in the game. Why do we need swimming? he'd ask. Because it'll look cool, was my reply. Sure enough, Carmack — doubtful yet accommodating — put the swim animation in and everyone went "Ahhh..." or "Ooooh..." He became a believer.

Another case was the jump animations. I was given one paltry frame to represent a recovery from jumping forward or backward (two separate animations). Characters in our game jump the equivalent of 20 feet in the air — certainly they needed to be able to recover from that kind of air time. So I used up five to ten frames for recovery, and it turned out much better.

And since there were two kinds of jumps, I decided to differentiate the backward one by making the character do a back flip as he jumped. Who cares that your view doesn't change during the flip? This is a first-person game, remember? Everything about your character is done for the enjoyment and benefit of your opponents. Why not give them a show?

The Moral of the Story

The character animation technology was but one of several changes we made for the art in QUAKE 3: ARENA. In fact, even though it's a QUAKE product, Q3A's code shares similarities with its esteemed predecessors strictly by virtue that it's 3D and from id Software. Carmack was continually refining and optimizing the technology in all aspects of the game up until the very end of development.

As for my part in the implementation of the animation changes, more lucid communication with the programmer on some issues would saved me some time and trouble. But it took trial and error — many, many iterations and more than a smidgen of perseverance — to make it work. That's what integrating new technology is about: sticking it out and always trying to find a better solution to the problem at hand. That's one of the things that makes this type of work so challenging and rewarding. ■

The Integrated PC: More Consumers with Fewer Choices

It's always nice to note the surge in demand that accompanies the latest 3D graphics or audio product release. It's a sign of the health of high-end technologies, and consumers' insatiable appetite for a better multimedia experience. Fashionable technologies also make it difficult for game developers to predict the amount of devotion they should lavish on "the next great thing." To add to developers' woes, there is the current growth in low-cost PC systems, in which graphics and audio functions are integrated into chipsets in the motherboard, and multimedia processes are passed off to more powerful CPUs that are left idling until software catches up. The question is, are game developers reducing their total available market (TAM) by not targeting the base level of performance that these systems offer?

Putting It All Together

Intel is the big cheese of integration. This year, the company formally announced that it was getting out of the discrete graphics chip business. In other words, it wasn't going to make separate graphics components. Some attributed the move to the lackluster performance and market share of the company's 740 graphics chip. Intel has never been shy to abandon or change tactics, products, and initiatives when it sees its core business of selling CPUs threatened. An integrated PC is one in which the graphics and audio components can be thrown into the logic chips that control the flow of data from the CPU to memory and various controllers. The result is fewer components required, and more money left on the table for the CPU.

The mainstay of integration for Intel is the 810 Whitney chipset and its latest update, the 810e. At the heart of the 810 chipset is a memory controller with built-in graphics technology, called the 82810 graphics memory controller hub (GMCH). It's a long moniker, but in short, the graphics portion of the 810 chipset has integrated hardware motion compensation for software DVD video, as well as a digital video output port enabling connections to a traditional television or flat-panel display. There is also an integrated Audio Codec 97 (AC-97) controller that allows software audio and modem functionality using the host CPU. There's a lot more electronics than that, but suffice it to say that there is a measure of 2D, 3D, and audio functionality built into the chipset that helps negate the need for extra peripherals — or so the thinking goes.

The 810e is aimed at the market for Pentium II and Pentium III chips, while the original 810 is aimed at the Celeron and Pentium II markets. The basic differences between the various flavors of the 810 are the amount of cache memory they allow, as well the way they interface to the PCI and IDE buses. As you go higher up the food chain, the graphics improve, the PCI slots are more numerous, and the bus performance gets faster.

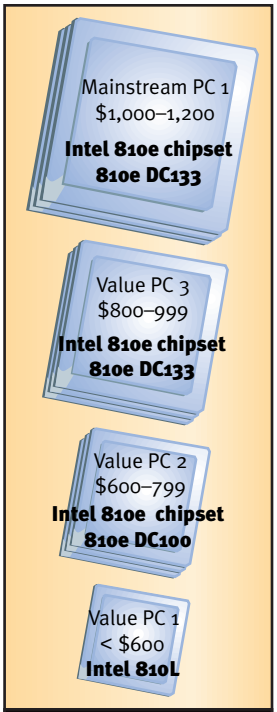


FIGURE 1: *Intel's overview of how its integrated products fit in to the PC market, by price segment.*

Integration's Impact on the Market

Intel isn't the only game in town, but its roadmap and gentle guidance, if you will, determine which way the PC industry goes. NeoMagic, a maker of graphics chips for laptops, has been creating integrated parts for some time, but in the desktop space there are companies such as Silicon Integrated Systems (SiS), Via, and a host of graphics chip vendors who all have the same idea for the low-cost PC market. There are no hard and fast rules about what integration means though, and the breadth of offerings will probably end up being a very confusing jumble of features. For instance, the SiS 620 chipset doesn't have all the features of the 810 (such as AC-97 support) and it is primarily appearing in very low-cost corporate PCs. What may be interesting down the road is something such as the Aladdin TNT2, a collaborative effort between Acer Laboratories Inc. (ALi) and Nvidia. Intel used the core graphics technologies of its 740 chipset, but Nvidia is look-

Omid Rahmat is the proprietor of Doodah Marketing, a digital media consulting firm. He also publishes research and market analysis notes on his web site at http://www.smokezine.com. He can be reached via e-mail at omid@compuserve.com.

FIGURE 2. *Shipments of integrated parts in desktop PCs in millions of units (source: Mercury Research).*

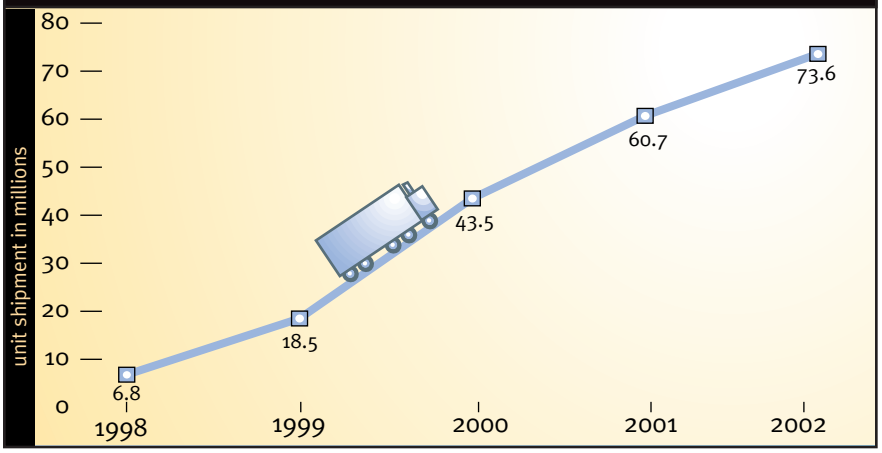
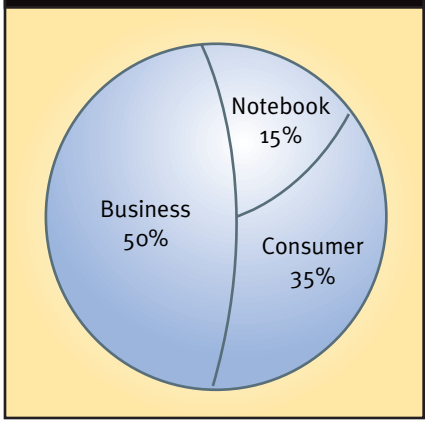


FIGURE 3. *Breakdown of integrated systems deployed by market segment (source: Mercury Research).*



ing to bring an altogether different level of performance to the low-cost PC market. And they aren't the only ones.

Intel's Whitney chipset holds about 80 percent of the market, according to Dean McCarron, principal analyst at Mercury Research, and provides graphics performance on par with a Matrox G200, or a little better than an ATI Rage Pro. The market for these products is still too young to judge the impact of integrated technologies, but Figures 1 and 2 may put things in perspective.

Obviously, the data suggest that integrated systems are going to take a significant slice of the PC market. In the consumer space, the primary market is for Internet-ready, low-cost PCs. The good news is that these types of PCs are attracting first-time PC buyers, very similar to the way the iMac has also drawn in a new crowd of computer users. These newcomers are going to want to do everything else their computers offer, including play games.

There is another dynamic of the integrated PC market that game developers might find interesting. According to McCarron, more audio add-ons are going into the consumer market than PCs sold. These add-ons are primarily high-end audio upgrades and 3D positional audio products such as Creative Labs' SoundBlaster Live! card. However, this is not yet happening on the graphics side, though it may still be too early to tell. On 810 motherboards you can't even get a graphics upgrade, and there is every indication that PC makers are finding it easier to upgrade their integrated-systems buyer to better quality audio than graphics.

So Where Does That Leave the Game Developer?

Integrated PCs are not going away. They are an expanding market and a very hot growth area for the PC industry. By ignoring this market, developers are reducing their TAM, which may be a conscious creative choice. After all, some may not be happy compromising the quality of their game graphics and audio. In time, it will certainly be a support issue for game developers because games are getting more sophisticated in their technology, not less so.

However, there is a new demographic entering the market as a result of these lower-cost integrated machine sales, the kind of user that can be lumped under the heading of mass market. The game industry doesn't quite understand the mass market yet, a fact quite evident by the seemingly endless parade of DEER HUNTER knockoffs. It will be a while before the game industry figures out how to target a user base that is highly motivated by the Internet, and less sanguine about traditional gaming. Furthermore, this is a demographic that is being weaned on low-cost hardware and budget software.

There is also a unique bundling opportunity for the game industry. The low-cost PC is going out with fewer new software titles in some cases, but often with a greater number of older titles. As such, it may be an ideal market for resurrecting the overall OEM market for games. In the past, I have addressed the OEM opportunities associated with the high end of the PC market, where games eat up the most CPU cycles and are sometimes the only justification for a powerful new system. The low-end users in the PC market, on the other hand, are more likely to be attracted by a bundle's value because of their price sensitivity. PC makers are beginning to realize that they need to sell their systems on more than just MIPS and hardware configurations. In short, there is a growing opportunity offered by integrated systems, but it comes at the price of a step back, certainly in graphics technology, and to some extent in audio technology, too. There is an opportunity for the game industry to increase its TAM, develop new OEM relationships, and bring new consumers into its fold. What remains to be done is to acquire a clear understanding of this demographic. That may not come for some time.

I am reminded of early days of the multimedia PC business, when hardware and software vendors misunderstood the needs of a market in which consumers were grabbing CD-ROM upgrade kits at $500 a pop, and bringing home $2,000 PCs by the cartload. The result was a mountain of useless multimedia CDs, and a crash in the price of some multimedia peripherals. The hardware and software industries need to do a better job with this new batch of consumers, and it all starts online. AOL is using Compuserve memberships and $400 rebates on consumer PCs to attract new users, and it seems to be working. The game industry needs to come up with new business models to deal with the mass market, or it may just end up being a passing fancy as well. ■


Back

THE CABAL

Valve's Design Process For Creating HALF-LIFE

By Ken Birdwell

While HALF-LIFE has seen resounding critical and financial success (winning over 50 Game of the Year awards and selling more than a million copies worldwide), few people realize that it didn't start out a winner — in fact, Valve's first attempt at the game had to be scrapped. It was mediocre at best, and suffered from the typical problems that plague far too many games. This article is about the teamwork – or "Cabal process" — that turned our initial, less than impressive version of HALF-LIFE into a groundbreaking success.

Paving the Way with Good Intentions

Our initial target release date was November 1997 — a year before the game actually shipped. This date would have given Valve a year to develop what was in essence a fancy QUAKE TC (Total Conversion — all new artwork, all new levels). By late September 1997, nearing the end of our original schedule, a whole lot of work had been done, but there was one major problem — the game wasn't any fun.

Yes, we had some cool monsters, but if you didn't fight them exactly the way we had planned they did really stupid things. We had some cool levels, but they didn't fit together well. We had some cool technology, but for the most part it only showed up in one or two spots. So you couldn't play the game all the way through, none of the levels tied together well, and there were serious technical problems with most of the game. There were some really wonderful individual pieces, but as a whole the game just wasn't working.

The obvious answer was to work a few more months, gloss over the worst of the problems and ship what we had. For companies who live and die at the whim of their publishers, this is usually the route taken — with predictable results. Since Valve is fairly independent, and since none of us believed that we were getting any closer to making a game we could all like, we couldn't see how a month or two would make any significant difference. At this point we had to make a very painful decision — we decided to start over and rework every stage of the game.

Ken is senior developer at Valve and has contributed to a wide range of projects in the last 15 years, most recently on animation and AI for HALF-LIFE. Previous projects include satellite networking, cryptography, 3D prosthetic design tools, 3D surface reconstruction, and in-circuit emulators. Oddly enough, Ken dropped out of studying EE to pursue a fine arts degree at The Evergreen State College, which he considers far more relevant to creative thinking than any silly differential equations class. You can reach him at kenb@valvesoftware.com.

Fortunately, the game had some things in it we liked. We set up a small group of people to take every silly idea, every cool trick, everything interesting that existed in any kind of working state somewhere in the game and put them into a single prototype level. When the level started to get fun, they added more variations of the fun things. If an idea wasn't fun, they cut it. When they needed a software feature, they simplified it until it was something that could be written in a few days. They all worked together on this one small level for a month while the rest of us basically did nothing. When they were done, we all played it. It was great. It was *Die Hard* meets *Evil Dead*. It was the vision. It was going to be our game. It was huge and scary and going to take a lot of work, but after seeing it we weren't going to be satisfied with anything less. All that we needed to do was to create about 100 more levels that were just as fun. No problem.

cabal \ka-'bal\ :to unite in a small party; to promote private views and interests by intrigue; to intrigue; to plot.

So, Tell Me About Your Childhood

The second step in the pre-cabal process was to analyze what was fun about our prototype level. The first theory we came up with was the theory of "experiential density" — the amount of "things" that happen to and are done by the player per unit of time and area of a map. Our goal was that, once active, the player never had to wait too long before the next stimulus, be it monster, special effect, plot point, action sequence, and so on. Since we couldn't really bring all these experiences to the player (a relentless series of them would just get tedious), all content is distance based, not time based, and no activities are started outside the player's control. If the players are in the mood for more action, all they need to do is move forward and within a few seconds something will happen.

The second theory we came up with is the theory of player acknowledgment. This means that the game world must acknowledge players every time they perform an action. For example, if they shoot their gun, the world needs to acknowledge it with something more permanent than just a sound — there should be some visual evidence that they've just fired their gun. We would have liked to put a hole through the wall, but for technical and game flow reasons we really couldn't do it. Instead we decided on "decals" — bullet nicks and explosion marks on all the surfaces, which serve as permanent records of the action. This also means that if the player pushes on something that should be pushable, the object shouldn't ignore them, it should move. If they whack on something with their crowbar that looks like it should break, it had better break. If they walk into a room with other characters, those characters should acknowledge them by at least looking at them, if not calling out their name. Our basic theory was that if the world ignores the player, the player won't care about the world.

A final theory was that the players should always blame themselves for failure. If the game kills them off with no warning, then players blame the game and start to dislike it. But if the game hints that danger is imminent, show players a way out and they die anyway, then they'll consider it a failure on their part; they've let the game down and they need to try a little harder. When they succeed, and the game rewards them with a little treat — scripted sequence, special effect, and so on — they'll feel good about themselves and about the game.

Conceptual artwork for a ceiling-mounted monster that was dangerous to both the player and the player's enemies.

Many of our scripted sequences were designed to give the player game-play clues as well as provide moments of sheer terror.

Secret Societies

Throughout the first 11 months of the project we searched for an official "game designer," — someone who could show up and make it all come together. We looked at hundreds of resumes and interviewed a lot of promising applicants, but no one we looked at had enough of the qualities we wanted for us to seriously consider them the overall godlike "game designer" that we were told we needed. In the end, we came to the conclusion that this ideal person didn't actually exist. Instead, we would create our own ideal by combining the strengths of a cross section of the company, putting them together in a group we called the "Cabal."

The goal of this group was to create a complete document that detailed all the levels and described major monster interactions, special effects, plot devices, and design standards. The Cabal was to work out when and how every monster, weapon, and NPC was

Tips For a Successful Cabal

- Include an expert from every functional area (programming, art, and so on). Arguing over an issue that no one at the meeting actually understands is a sure way to waste everyone's time.
- Write down everything. Brainstorming is fine during the meetings, but unless it's all written down, your best ideas will be forgotten within days. The goal is to end up with a document that captures as much as is reasonable about your game, and more importantly answers questions about what people need to work on.
- Not all ideas are good. These include yours. If you have a "great idea" that everyone thinks is stupid, don't push it. The others will also have stupid ideas. If you're pushy about yours, they'll be pushy about theirs and you're just going to get into an impasse. If the idea is really good, maybe it's just in the wrong place. Bring it up later. You're going to be designing about 30 hours of game play; if you really want it in it'll probably fit somewhere else. Maybe they'll like it next month.
- Only plan for technical things that either already work, or that you're sure will work within a reasonable time before play testing. Don't count on anything that won't be ready until just before you ship. Yes, it's fun to dream about cool technology, but there's no point in designing the game around elements that may never be finished, or not polished enough to ship. If it's not going to happen, get rid of it, the earlier the better.
- Avoid all one-shot technical elements. Anything that requires engineering work must be used in more than one spot in the game. Engineers are really slow. It takes them months to get anything done. If what they do is only used once, it's a waste of a limited resource. Their main goal should always be to create tools and features that can be used everywhere. If they can spend a month and make everyone more productive, then it's a win. If they spend a week for ten seconds of game play, it's a waste.

to be introduced, what skills we expected the player to have, and how we were going to teach them those skills. As daunting as that sounds, this is exactly what we did. We consider the Cabal process to have been wildly successful, and one of the key reasons for HALF-LIFE's success.

Cabal meetings were semi-structured brainstorming sessions usually dedicated to a specific area of the game. During each session, one person was assigned the job of recording and writing up the design, and another was assigned to draw pictures explaining the layout and other details. A Cabal session would typically consist of a few days coming up with a mix of high level concepts for the given area, as well as specific events that sounded fun.

Once enough ideas were generated, they would be reorganized into a rough storyline and chronology. Once this was all worked out, a description and rough sketch of the geometry would be created and labeled with all the key events and where they should take place. We knew what we wanted for some areas of the game from the very start, but other areas stayed as "outdoors" or "something with a big monster" for quite some time. Other areas were created without a specific spot in the game. These designs would sit in limbo for a few weeks until either it became clear that they weren't going to fit, or that perhaps they would make a good segue between two other areas. Other portions were created to highlight a specific technology feature, or simply to give the game a reason to include a cool piece of geometry that had been created during a pre-cabal experiment. Oddly enough, when trying to match these artificial constants, we would often create our best work. We eventually got into the habit of placing a number of unrelated requirements into each area then doing our best to come up with a rational way to fit them together. Often, by the end of the session we would find that the initial idea wasn't nearly as interesting as all the pieces we built around it, and the structure we had designed to explain it actually worked better without that initial idea.

During Cabal sessions, everyone contributed but we found that not everyone contributed everyday. The meetings were grueling, and we came to almost expect that about half of the group would find themselves sitting through two or three meetings with no ideas at all, then suddenly see a direction that no one else saw and be the main contributor for the remainder of the week. Why this happened was unclear, but it became important to have at least five or six people in each meeting so that the meetings wouldn't stall out from lack of input.

The Cabal met four days a week, six hours a day for five months straight, and then on and off until the end of the project. The meetings were only six hours a day, because after six hours everyone was emotionally and physically drained. The people involved weren't really able to do any other work during that time, other than read e-mail and write up their daily notes.

The initial Cabal group consisted of three engineers, a level designer, a writer, and an animator. This represented all the major groups at Valve and all aspects of the project and was initially weighted towards people with the most product experience (though not necessarily game experience). The Cabal consisted only of people that had actual shipping components in the game; there were no dedicated designers. Every member of the Cabal was someone with the responsibility of actually doing the work that their design specified, or at least had the ability to do it if need be.

The first few months of the Cabal process were somewhat nerve wracking for those outside the process. It wasn't clear that egos could be suppressed enough to get anything done, or that a vision of the game filtered through a large number of people would be anything other than bland. As it turned out, the opposite was true; the people involved were tired of working in isolation and were energized by the collaborative process, and the resulting designs had a consistent level of polish and depth that hadn't been seen before.

Internally, once the success of the Cabal process was obvious, mini-Cabals were formed to come up with answers to a variety of design problems. These mini-Cabals would typically include people most effected by the decision, as well as try to include people completely outside the problem being addressed in order to keep a fresh perspective on things. We also kept mem-

The team explored a variety of visual metaphors that resulted in some very unique and effective opponents.

bership in the initial Cabal somewhat flexible and we quickly started to rotate people through the process every month or so, always including a few people from the last time, and always making sure we had a cross section of the company. This helped to prevent burn out, and ensured that everyone involved in the process had experience using the results of Cabal decisions.

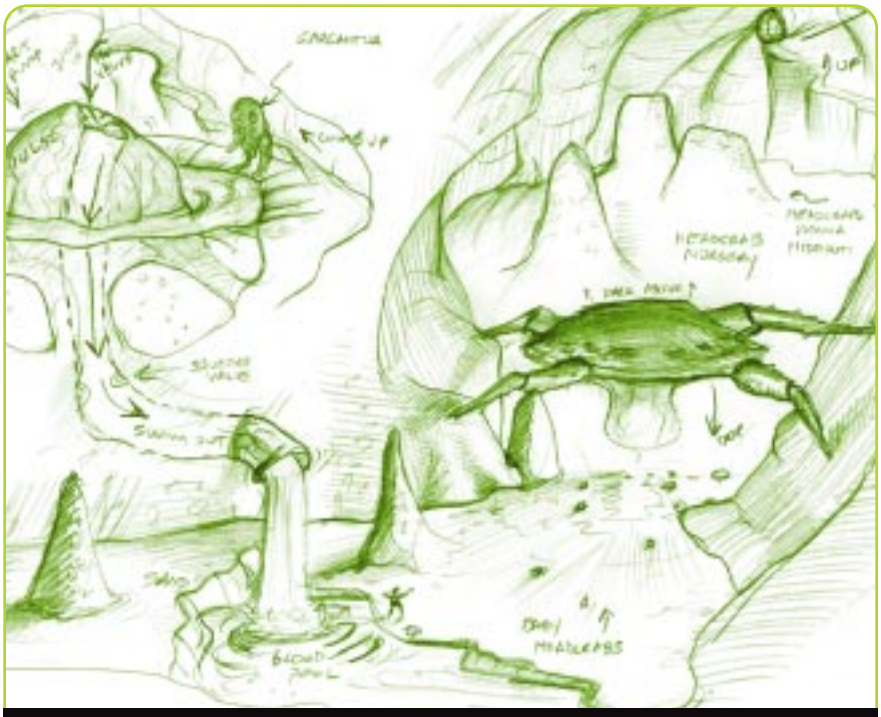
The final result was a document of more than 200 pages detailing everything in the game from how high buttons should be to what time of the day it was in any given level. It included rough drawings of all the levels, as well as work items listing any new technology, sounds, or animations that those levels would require.

We also ended up assigning one person to follow the entire story line and to maintain the entire document. With a design as large as a 30-hour movie, we ended up creating more detail than could be dealt with on a casual or part-time basis. We found that having a professional writer on staff was key to this process. Besides being able to add personality to all our characters, his ability to keep track of thematic structures, plot twists, pacing, and consistency was invaluable.

Pearls Before Swine

By the second month of the Cabal, we (the "swine") had enough of the game design to begin development on several areas. By the third month, we had enough put together to begin play testing.

A play-test session consists of one outside volunteer (Sierra, our publisher, pulled play-testers from local people who had sent in product registration



It's important to include information on the intended path through the level, as well as rough geometry and character placement.

Play-Testing Tips

- Watch your play-testers and let them do what they want. If they keep trying to do something silly, don't get upset — figure out why they want to do it and how to accommodate them. We originally added breakable crates to the game simply as a technological feature that we wanted to show off. There was nothing in them, the crates just broke. We thought games with secrets hidden in crates were lame. After the tenth play-tester in a row went through and painstakingly broke every single crate in the entire game and never got any reward but just kept doing it (because they just knew if they just tried long enough they'd get the hidden reward) we caved. We went back and redesigned levels to have crates with goodies in them, and a reason to need those goodies. It was a lot of work, but the remaining play-testers were a lot happier.
- Your game is too hard. It just is. You're most likely an expert player at the game you're developing and it's doubtful that there are more than a handful of players in the world who are better than you. You don't need to care about them, you need to care about the 99.9 percent of the players who aren't as good as you. Some of those players will be really, really bad. Tough — they've paid their $50 and they deserve to be entertained. Make the difficult level something you can play without too much trouble. Make the easy level so easy that you can't imagine anyone not being able to win on it. Then, make it a bit easier. If you get lucky, half your players will be able to finish.

This creature was initially designed as a friendly character, but play-testing revealed players' tendencies to shoot first and ask questions later.

cards for other games) playing the game for two hours. Sitting immediately behind them would be one person from the Cabal session that worked on that area of the game, as well as the level designer who was currently the "primary" on the level being tested. Occasionally, this would also include an engineer if new AI needed to be tested.

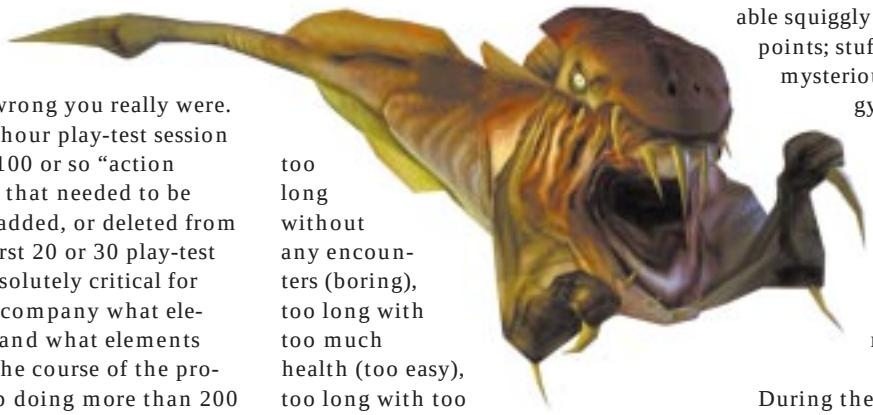
Other than starting the game for them and resetting it if it crashed, the observers from Valve were not allowed to say anything. They had to sit there quietly taking notes, and were not allowed to give any hints or suggestions. Nothing is quite so humbling as being forced to watch in silence as some poor play-tester stumbles around your level for 20 minutes, unable to figure out the "obvious" answer that you now realize is completely arbitrary and impossible to figure out.

Letting players see other characters make mistakes that they'll need to avoid is an effective way to explain your puzzles and add tension and entertainment value.

This was also a sure way to settle any design arguments. It became obvious that any personal opinion you had given really didn't mean anything, at least not until the next play-test session. Just because you were sure something was going to be fun didn't make it so; the play-testers could still show up and demonstrate just how wrong you really were.

A typical two-hour play-test session would result in 100 or so "action items" — things that needed to be fixed, changed, added, or deleted from the game. The first 20 or 30 play-test sessions were absolutely critical for teaching us as a company what elements were fun and what elements were not. Over the course of the project we ended up doing more than 200 play-test sessions, about half of them with repeat players. The feedback from the sessions was worked back into the Cabal process, allowing us to preemptively remove designs that didn't work well, as well as elaborate on designs that did.

Toward the middle of the project, once the major elements were in place and the game could be played most of the way through, it became mostly a matter of fine-tuning. To do this, we added basic instrumentation to the game, automatically recording the player's position, health, weapons, time, and any major activities such as saving the game, dying, being hurt, solving a puzzle, fighting a monster, and so on. We then took the results from a number of sessions and graphed them together to find any areas where there were problems. These included areas where the player spent too long without any encounters (boring), too long with too much health (too easy), too long with too little health (too hard), all of which gave us a good idea as to where they were likely to die and which positions would be best for adding goodies.



Another thing that helped with debugging was making the "save game" format compatible between the different versions of the engine. Since we automatically saved the game at regular intervals, if the play-testers crashed the game we would usually have something not too far from where they encountered the bug. Since these files would even work if the code base they were testing was several versions old, it made normally rare and hard to duplicate bugs relatively easy to find and fix. Our save game format allowed us to add data, delete data, add and delete code (we even supported function pointers) at will, without breaking anything. This also allowed us to make some fairly major changes after we shipped the game without interfering with any of our players' hard-won saved games.

No Good Deed Goes Unpunished

Until the Cabal process got underway, technology was added to HALF-LIFE freely. It was assumed that "if we build it, they will come," meaning that any new technology would just naturally find a creative use by the content creation folks. A prime example of this fallacy was our "beam" effect, basically a technique for doing highly tunable squiggly glowing lines between two points; stuff like lightning, lasers, and mysterious glowing beams of energy. It was added to the engine, the parameters were exposed, and an e-mail was sent out explaining it. The result was … nothing. After two months only one level designer had put it in a map. Engineering was baffled.

During the Cabal process, we realized that although the level designers knew of the feature, they really had no clear idea of what it was for. The parameters were all very cryptic, and the wrong combinations would cause the beams to have very ugly-looking effects. There were no decent textures to apply to them, and setting them up was a bit of a mystery. It became very

Second-Order Effects

Early in the development of HALF-LIFE, skeletal animation was added solely for compression reasons; we needed to be able to store a lot more frames of animation in fewer bytes than were required by vertex animation. The DSP effects and the reworking of the sound system, were added solely for the ability to add echo and other single-processing effects. Adding a mouth to the various humanoid models was solely for the humor value of having the scientists scream while playing specific scripted sequences. Each by itself was considered complete.

Then the animator who added the mouths asked the programmer who wrote the DSP code if there was any way to make the mouth move when the character talked. He was told it would be easy to know when the character was talking but that there wasn't any way to automatically move the jaw. He then asked the programmer who wrote the skeletal animation code if there was any way to make the mouth move when the character talked. He was told it would be easy to move the jaw, but that there wasn't really any way to tell when they were talking. Independently, both programmers told him that it was impossible to do and to quit bugging them about it. It wasn't until several weeks later when the programmers were sitting around commenting on what new impossible feature the artists wanted that they realized that the part neither one of them understood was in fact trivial. About an hour later, the characters in the game could talk.

Of course, that wasn't the end of it. Since the Cabal had just started, and since we now had a solution to the problem of how to explain to the player what was going on — now other characters could tell them — it was time to rework our existing designs. With the ability to talk to the player, we now needed some character in the game that the player would actually want to talk to instead of just killing right away. We decided that the trigger-happy security guard monster — originally designed as an easy version of the soldiers — should instead be a supporting character. The scientists also went from being fairly ambiguous — good or evil, we had them both ways — to being definitely on the player's side. It also meant that instead of claiming we had a story because we included a bunch of prose in a README.TXT file somewhere, we could have a real author do real dialog with real (within technical limits) characters and use real storytelling techniques.

We've come to the conclusion that major ideas can come from anyone, and that most technologies have hidden features in them that will be discovered only after the initial technology is in place. For this reason, it's critical to include everyone in the design process and create a mechanism to feed these second-order effects back into the development process.

clear the technology itself was only a small part of the work and integration, training, and follow-through were absolutely necessary to make the technology useful to the game. Writing the code was typically less than half the problem.

Square Pegs

Practically speaking, not everyone is suited for the kind of group design activity we performed in the Cabal, at least not initially. People with strong personalities, people with poor verbal skills, or people who just don't like creating in a group setting shouldn't be forced into it. We weighted our groups heavily toward people with a lot of group design experience, well ahead of game design experience. Even so, in the end almost everyone was in a Cabal of one sort or another, and as we got more comfortable with this process and started getting really good results it was easier to integrate the more reluctant members. For current projects, such as TEAM FORTRESS 2, the Cabal groups are made up of 12 or more people, and rarely fewer than eight. The meetings ended up being shorter, and they also ended up spreading ideas around a lot quicker, but I'm not sure I'd recommend that size of group initially.

Just about everything in HALF-LIFE was designed by a Cabal. This at first seemed to add a bit of overhead to everything, but it had the important characteristic of getting everyone involved in the creation process who were personally invested in the design. Once everyone becomes invested in the design as a whole, it stops being separate pieces owned by a single person and instead the entire game design becomes "ours."

This "ours" idea extended to all levels. Almost every level in the game ended up being edited by at least three different level designers at some point in its development and some levels were touched by everyone. Though all the level designers were good at almost everything, each found they enjoyed some aspect of level design more than other aspects. One would do the geometry, one would do monster and AI placement, our texture artist would step in and do a texturing pass, and then one would finish up with a lighting pass, often switching roles when needed due to scheduling conflicts. This became critical toward the end of the project when people finished at different times. If a play-test session revealed something that needed to be changed, any available level designer could make the changes without the game getting bottlenecked by needing any specific individual.

This idea also extended to all code, textures, models, animations, sounds, and so on. All were under source control and any individual was able to synch up to the sources and make whatever changes were necessary. With a little bit of self–control, this isn't as random as it sounds. It had the added benefit in that it was fairly easy to get a daily record of exactly what was changed and by whom. We would then feed this information back into the play-test cycles, only testing what had changed, as well as helping project scheduling by being able to monitor the changes and get a pretty good estimate of the stability and completeness of any one component. This also allowed us to systematically add features throughout the process with minimal impact. Once the technical portion was completed, the engineer assigned to the feature was able to synch to all the source artwork and rebuild any and all files (models, textures, levels, and so on) affected by the change.

The skeletal system allowed the team to change its monster appearance throughout the development process with minimal impact on existing animations and AI.

By placing traditional combat action in more challenging environments we were able to intensify the feeling of tension and suspense.

The Workers Control the Means of Production

Even with all emphasis on group activity, most of the major features of HALF-LIFE still only happened through individual initiative. Everyone had different ideas as to what exactly the game should look like, or at least what features we just had to do. The Cabal process gave these ideas a place to be heard, and since it was accepted that design ideas can come from anyone, it gave people as much authority as they wanted to take. If the idea required someone other than the inventor to actually do the work, or if the idea had impact on other areas of the game, they would need to start a Cabal and try to convince the other key people involved that their idea was worth the effort. At the start of the project, this was pretty easy as most everyone wildly underestimated the total amount of work that needed to be done, but toward the middle and end of the project the more disruptive decisions tended to get harder and harder to push through. It also helped filter out all design changes except for the ones with the most player impact for the least development work.

Through constant cycle of play-testing, feedback, review, and editing, the Cabal process was also key in removing portions of the game that didn't meet the quality standards we wanted, regardless of the level of emotional attachment the specific creator may have had to the work. This was one of the more initially contentious aspects of the Cabal process, but perhaps one of the more important. By its very nature, the Cabal process avoided most of the personal conflicts inherent in other more hierarchical organizations. Since problems were identified in a relatively objective manner of play-testing, and since their solutions were arrived at by consensus or at least by an individual peer, then an authority that everyone could rebel against just didn't exist.

On a day-to-day basis, the level of detail supplied in even a 200-page design document is vague at best. It doesn't answer the 1,001 specific details that each area requires, or the countless creative details that are part of everyday development. Any design document is really nothing more than a framework to work from and something to improve the likelihood that work from multiple people will fit together in a seamless fashion. It's the Cabal process that helped spread around all the big picture ideas that didn't make it into any document — things that are critical to the feel of the game, but too nebulous to put into words. It also helps maximize individual strengths and minimize individual weaknesses and sets up a framework that allows individuals to influence as much of the game as possible. In HALF-LIFE, it was the rare area

Technology Integration Tips

- Work directly with the artists — level designers, animators, and so on — for however long it takes to get the technology into a sample portion of the game. Sit in their office. Hang out. Watch what they do. Watch what they try. At this point, you will probably need to simplify, enhance, and/or document the interface better.
- Have the artist demo the technology to the rest of their group. When it comes from outside, it'll be viewed with a certain amount of skepticism. When it comes from within, it'll be viewed as a new tool.
- Reintegrate the technology into the world. You've spent the time adding the new technology, but if the player never sees it, then all your work will be wasted. Make sure it gets into the game design document. Maybe it should be used instead of other effects. Maybe it can be used to enhance an existing portion. Make sure it gets into as much of the game as possible.
- Keep track of how the technology is used. After a few months you will probably notice that the technology is being used to do things you never considered. Analyze how it's being used and look for improvements or new tools that can make it even better. You'll need to keep this up for the entire project.

of the game that didn't include the direct work of more than ten different people, usually all within the same frame.

In order for highly hierarchical organizations to be effective, they require one person who understands everyone else's work at least as well as the individuals doing the work, and other people who are willing to be subordinates yet are still good enough to actually implement the design. Given the complexity of most top game titles, this just isn't practical — if you were good enough to do the job, why would you want to be a flunky? On the other hand, completely unstructured organizations suffer from lack of information and control — if everyone just does their own thing, the odds that it'll all fit together in the end are somewhere around zero.

The first incarnation of the game's main character, now known affectionately as "Ivan the Space Biker."

At Valve, we're very happy with the results of our Cabal process. Of course, we still suffer from being overly ambitious and having, at times, wildly unrealistic expectations, but these eventually get straightened out and the Cabal process is very good about coming up with the optimal compromise. Given how badly we failed initially, and how much the final game exceeded our individual expectations, even our most initially reluctant person is now a staunch supporter of the process. ■

Placing the player in a soldier-vs.-alien conflict helped reinforce the illusion of an active environment, and let Valve show off its combat AI with minimal risk to the player.

Making a Break from Fragile Code Design

by James Boer

Is maintaining your game libraries becoming more and more of a challenge? Is your game's code base becoming increasingly fragile? This isn't an uncommon phenomenon. We've all heard about projects that were cancelled due to unmaintainable code and runaway bug counts. Perhaps you've even been part of such a project. The key to avoiding this kind of catastrophe lies in good code design and a willingness to set a certain level of discipline in your team's coding style.

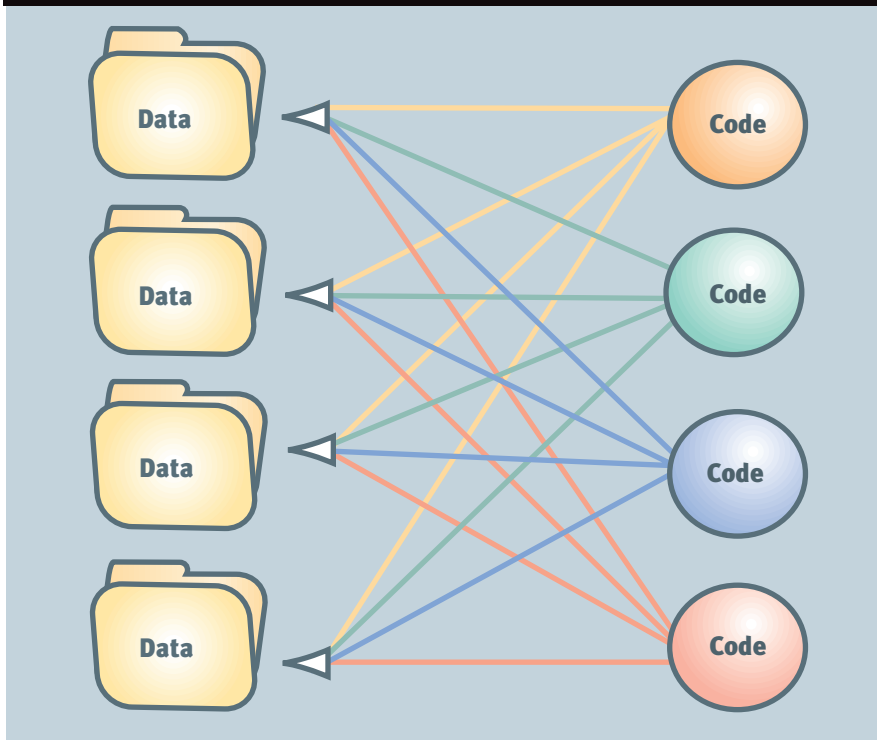
Game development offers a unique challenge to programmers because of its combination of cutting-edge technology and traditional software development challenges. Often, traditional solutions to program design tend to fall short because of the rapid evolution of game development technologies. For instance, Microsoft's DirectX libraries constantly evolve, and each new version has new features and takes advantage of new hardware and technologies. But trying to ensure that your programs will be able to use the newest versions of DirectX without having to rewrite lots of code is made more difficult in the process. The ability to design upgradability into your software libraries without sacrificing stability is a rare art today, but it doesn't have to be so. To combat fragile code, this article examines some traditional approaches and discusses the benefits and limitations of these methods.

Ever-Increasing Complexity

The day of the free-coding, seat-of-your-pants, I-don't-believe-in-designing-code-before-I-write-it programmer is fast drawing to an end. Today's top games have become as sophisticated as mainstream commercial applications, and many companies feel the pressures that inspired more efficient design methodologies such as object-oriented programming and the concepts of design patterns. As programs get larger and more complex, the number of interdependencies between code modules tends to increase at an exponential rate, making code reuse or redesign more difficult. Companies that do not feel compelled to design their software effectively will eventually be surpassed in the marketplace by those who do, producing more bug-free code with a shorter turnaround time.

James Boer is the designer and one of the programmers who brought you DEER HUNTER and DEER HUNTER II (and he still has never gone hunting). Other game credits include ROCKY MOUNTAIN TROPHY HUNTER, PRO BASS FISHING, MICROSOFT BASEBALL 2000, and MICROSOFT INTERNET HEARTS. He currently is working at WizBang! Software in Seattle, Wash., on a yet unannounced product, and can be reached for comments, questions, or general advice about virtual deer hunting at jbsys@compuserve.com.

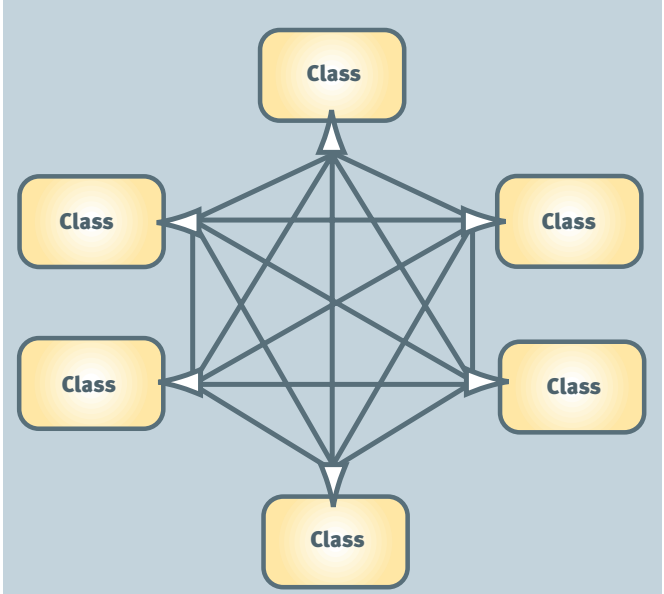
FIGURE 1. *Cohesion between code and data.*



The Pitfalls of Procedural Programming

In procedural programming, data is a distinctly separate entity from the code that operates on that data (Figure 1). This creates a situation in which numerous modules of code might be looking at a single chunk of data. As the number of code and data modules increases, you tend to see data and code modules become more dependent on each other. This web of dependencies, often called "code cohesion," makes it increasingly difficult to alter any one component without adversely affecting the others.

FIGURE 2. *In OOP, different code modules can become too dependent upon each other.*



When game code becomes too cohesive, the entire system must either be redesigned or discarded due to its inherent lack of flexibility. This process is known as refactoring, and it is an inevitable part of any software development cycle. The problem is not how to avoid the need to refactor your software. This is an unrealistic goal, as no developer can anticipate changing future requirements to the degree that it will never need to be redesigned at a fundamental level. However, you can strive to maximize the effective lifespan of code. This goal, achieved through the use of reusable components which encapsulate both code and data, was what object-oriented programming (OOP) promised to deliver. Unfortunately, OOP can't provide all the answers on its own. (See sidebar, "Code Cohesion," p. 58)

Shortcomings of OOP

OOP encapsulates code and data into a single module, which in essence, gives the data a code interface. This means that the underlying data can be changed without modifying the interface to access that data, which provides an additional level of protection when requirements change. However, just as dependencies between code and data modules can escalate to an unmanageable level, dependencies among different code modules can also create an unmanageable level of cohesion (Figure 2).

Using OOP techniques has simply delayed the progression of the cohesion by combining code and data modules. As programs grow in complexity, however, the code dependencies still increase to an undesirable level. To move away from the abstract a bit, let's take a look at a real problem and see how a real solution was implemented.

Real-Life Problem: A DirectX Upgrade

The issue of using DirectX libraries effectively is something most PC game developers have had to face. Although these libraries attempt to remain backward-compatible with the interfaces of previous versions, there are times when compatibility must be broken in order to introduce new features. It is at this point that your existing code becomes incompatible with the new libraries. The typical OOP solution to an evolving interface is the creation of a thin wrapper class for the library, which uses inline functions to shield the game programmer from the more mundane tasks involved in using the library.

Code Cohesion

What is code cohesion, what causes it, and exactly why is it such a bad thing? Cohesion is the result of too many parts of a program becoming dependent on each other. Of course, some cohesion happens by necessity in any program written, but the degree of cohesion is what's important. If the amount of cohesion remains low, it is easier to replace any particular subsystem of code because the impact on other portions of the code will be minimal. If, on the other hand, the code is highly dependent upon specific interfaces and data structures from many other parts of the program, it becomes impossible to change without affecting a vast number of other systems.

Cohesion is caused by a number of things, but there are some more obvious coding practices to watch out for. Here are some programming practices which, unless used cautiously, can lead to highly cohesive code:

1. **USING GLOBAL VARIABLES AND DATA STRUCTURES.** There are some instances where globals might be necessary, but think long and hard before using them. More often than not, these types of items can be encapsulated in a class with little ill effect. Structs, unless used exclusively inside a particular class, should be used with caution. Wrapping data and functionality into a single class is usually a safer approach. The biggest problem with using a struct to store common data is that it's difficult to manage and track who is actually accessing the data and when they are doing it.

2. **INCORRECT SCALING OF CLASSES AT DESIGN TIME.** It's a common mistake for programmers to design their classes too small and specific, resulting in many smaller interdependent objects, or too large, resulting in complex, monolithic objects that attempt to do far too much. It's important to design and use classes at both ends of the size scale, both to keep code modules small and readable, yet also to keep the overall design relatively understandable.

3. **ALLOWING MANY SMALL OBJECTS IN DIFFERENT SUBSECTIONS OF THE CODE TO COMMUNICATE DIRECTLY WITH EACH OTHER.** Although this is sometimes unavoidable, you should always try to make use of a high-level interface to communicate between subsections of code. Inline function can alleviate most concerns about addition overhead imposed by the additional interface layer. If direct communication between smaller objects absolutely must occur, try to utilize some communication abstraction if possible as a go-between, such as a message object.

Alternatively, here's three techniques you can use to ensure you're code doesn't become too highly cohesive.

1. **MAKE GOOD USE OF LIBRARIES.** If you can separate your code from the application and place it into a library, you've established a natural separation that cannot easily be breached. Objects in the library have to remain relatively independent of the code in the application.

2. **FOCUS ON THE "BLACK BOX" INTERFACE OF YOUR OBJECTS.** Your objects should be performing tasks under the hood, without other programmers having to worry about exactly how the tasks are being implemented. This means that you should keep the public function available for use, make sure that it's easily implemented, and document it well (including good comments within the code). Keep your helper functions private or protected. If you find the number of functions growing extremely large, perhaps it's time for a higher-level object to take over the task, or to consolidate a number of the smaller functions into larger, simpler-to-use function.

3. **HAVE A SOLID DESIGN STRATEGY BEFORE CODING STARTS.** This sounds a bit too simplistic to be a practical rule, but it's an important one nonetheless. Don't resign yourself to the illusion that you'll go back later and "do it right." In my experience, there's rarely the time or inclination to do that. Once code is written and "working," it's hard to convince someone that you need to go back and spend more time to rewrite code that essentially does exactly the same thing.

For example, blitting from surface to surface in DirectDraw requires the programmer to fill out a number of parameters in the `IDirectDrawSurface4::Blt()` method, including source and destination rectangles, a source surface, flags, and a blit structure. The Component Object Model (COM) structure, on which DirectX is based, provides much in the way of standardization, but unfortunately it has no concept of default parameters and function overloading. So you might consider writing wrapper `Blt()` functions which are easier to use. You also might have several overloaded `Blt()` functions for different situations. Class wrapping like this makes an API easier to work with and also helps shield the application from changes in (for this case) the DirectDraw interface. If function parameters within `Blt()` change, a programmer must only make changes inside the wrapper functions instead of throughout the game.

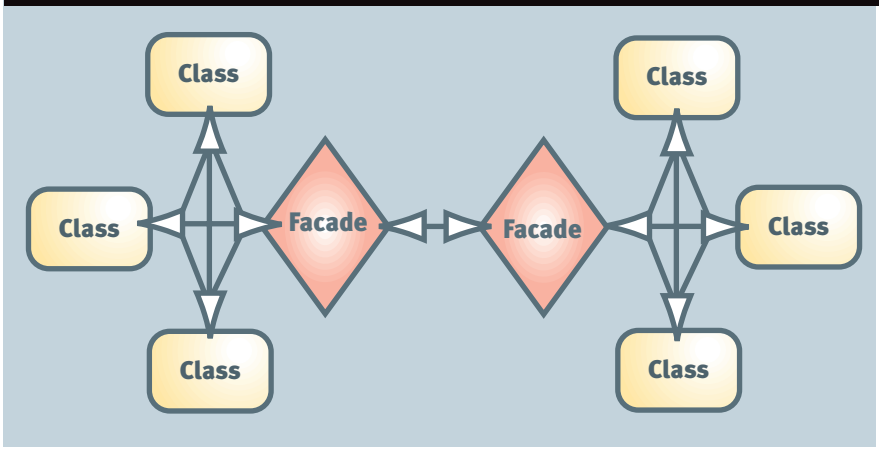
Unfortunately, this doesn't provide protection from major changes in an API's functionality. For instance, in DirectX 6.0, Microsoft introduced Direct3D vertex buffers as an alternative to execute buffers. This type of change typically has more far-reaching effects in an application than a simple change in a function's parameters. So some solution must be found to protect code subsystems from major changes in one or more basic components of another system.

The Facade Pattern: Half a Solution

The facade pattern provides a solution to the seemingly inevitable cohesion between different modules of code in a program. The basic premise of the facade pattern is this: all code must be logically divided into discreet modules, and these modules should, whenever possible, communicate to each other through the use of an interface known as a facade. The facade is a high-level interface to the functionality of a complete subsystem of objects which work together to perform some related task. There will be times when lower-level functionality of a system must be accessed directly, but often the facade can take care of higher-level functions more easily than another solution.

These are not radical or even new concepts. Most programmers have seen the benefits of some sort of manager class that is used as the interface to the rest of the program. Using a system such as this helps to insulate the remainder of a program when the implementation of a subsystem radically changes under the hood (Figure 3).

FIGURE 3. *Visual representation of communication between subsystems using facades.*



The Virtual Manager Alternative

Although the facade works well, it does present some serious limitations. Let's assume we wish to build a facade over every one of our major subsystems, which we've split into separate libraries. This means that each major library has a facade to handle communication to and management of the subsystem, creating a cleaner interface between each subsystem and with the application. However, most applications require some customization of their libraries to one degree or another as the requirements of the application evolve over time and new features are added. If more than one application uses the same set of libraries, then the customizations must occur at the application level, instead of the library level. The problem then boils down to: How can we customize the libraries at the application level and still allow other libraries to use those customizations?

A while back, we discovered a problem when working on specific library modules that needed to access code in other libraries we had written. The UI library, for example, required the use of our Graphics and Input libraries. How can we ensure that the UI library can access those other libraries' functionality, even though an application may wish to extend the Graphics or Input libraries by deriving new classes from them at the application level?

The answer lies in the clever use of a static pointer. By instantiating a class at the application level and passing the pointer of that object back to the class to store as a static member, we give every piece of code, whether at the library or application level, the ability to access the object that the application has created. This object may be an instance of the class which resides in the library, or it may actually be an instance of a class derived from it. From the other library module's point of view, it makes no difference. The solution is simple and elegant, and provides a good deal of flexibility. Here's what the code looks like:

```
// sample code showing a generic manager
// class
class ManagerBase
{
   public:
      ManagerBase();
      virtual ~ManagerBase();

      ManagerBase* GetBase()
      { return m_pBaseObj; }

      void SetBase(ManagerBase* pObj)
      {  m_pBaseObj = pObj; }

   protected:
      static ManagerBase* m_pBaseObj;
};
```

We see a simple mechanism for storing a single object in its own static member. Access to the object is achieved through the `GetBase()` function. However, in order to avoid having to type in:

```
ManagerBase::GetBase()->DoSomeFunction();
```

every time we want to call a function, we can simplify the call through the strategic use of an inline function.

```
inline ManagerBase* Manager()
{ return ManagerBase::GetBase(); }
```

Design Patterns

There's been a lot of talk about design patterns in the world of object-oriented programming (OOP). In a nutshell, a design pattern is a simply way of describing the general object-oriented form of a solution to a specific type of programming problem. If this sounds a little vague or abstract to you, it should. Patterns only give you a blueprint of object interaction to model your own classes on. After that, it's up to you to build the specific classes that solve whatever problem you're facing. What the pattern gives you is a strategy when deciding which objects to use, and how they should work together.

Effective object-oriented design is one of the more elusive skills a programmer can master. C++ is really not all that hard to learn at the basic language level. After all, one doesn't necessarily have to use templates, exceptions, and other fancy language features. However, it has a reputation of being a difficult language to master due in part to the complex art of designing classes that are clean, efficient, and expandable. Patterns can help shorten this learning curve by describing common programming problems, then showing how a class or hierarchy of classes can be used to solve the problem with proven, time-honored class designs.

In addition to allowing programmers to see practical object-oriented solutions to programming tasks that they may encounter, patterns also give programmers a common frame of reference when talking about programming solutions by naming the patterns. If I describe a class utilizing a Factory pattern (a Factory is responsible for creating objects and returning interfaces to them), anyone who understands what a Factory pattern is automatically can relate to the general nature and purpose of the class that I'm attempting to describe.

This function is placed just below the class definition in the header file, and should be used to access the object. Code calling the manager then looks like this:

```
Manager()->DoSomeFunction();
```

This looks quite a bit more friendly, and because the function is inline, requires no additional overhead.

You'll notice that the class was named `ManagerBase` instead of `Manager`. This was done with the foresight of knowing that the inline accessor function, not the object, should have the name most closely associated with the actual functionality of the object. For instance, with a sound manager, you should name the object `SoundManagerBase`, which leaves you free to name the accessor function `SoundManager()`.

The application is responsible for creating the manager object and passing back to the class for storage, like this:

```
// application initialization code

ManagerBase* pBase = new ManagerBase;
if(!pBase)
   return Error;
ManagerBase::SetBase(pBase)

// now we're ready to call the manager
// functions
Manager()->DoSomeFunction();
```

Because the application is responsible for creating and setting the object, it's possible to try to access the manager without having initialized it. This will result in an access violation, so it is important to guard against this. Ideally, the object should be created at the beginning of the application and destroyed at the end, which will guard against this type of error. If for any reason parts of your code might try to access a manager before it has been created, it is your responsibility to wrap that code in an `if` statement which checks to see that a valid object exists before accessing it. This is the biggest drawback to this system, but with careful planning it doesn't have to be a problem.

A Real-Life Solution

For an example of a practical solution, let's look at a sound manager, as it's a relatively simple subsystem. Most sound managers consist of code that manages a list of sounds, and can load, unload, play, stop, or set properties of any of these sounds on demand. Most applications should be able to use this set of functions without much alteration.

At the application level, however, the benefits of overriding the manager class become quickly apparent. For instance, you may wish to add the capabilities of associating some other types of data with some sounds. For a technology demo of an upcoming product we were showing to publishers, we associated a lip-synch data file with voice-over recordings. After the voice-over sounds were loaded, we could retrieve the associated data file as well.

A class, which we'll call `AppSoundManagerObj`, is derived from `SoundManagerBase`. In addition to deriving the class from the base class, we'll also create a new accessor function, like this:

```
inline AppSoundManagerObj * AppSoundManager()
{ return ((AppSoundManagerObj
*)SoundManager()); }
```

This new accessor gives us a pointer to the new `AppSoundManagerObj` interface, which of course also includes the interface to the original base manager class.

In addition to adding new members to the derived class, the ability to derive means that we also gain the ability to change default behavior of the manager classes by overriding virtual functions. This effectively lets us change the behavior of the manager even when it is called from other libraries.

Migrating Library Code

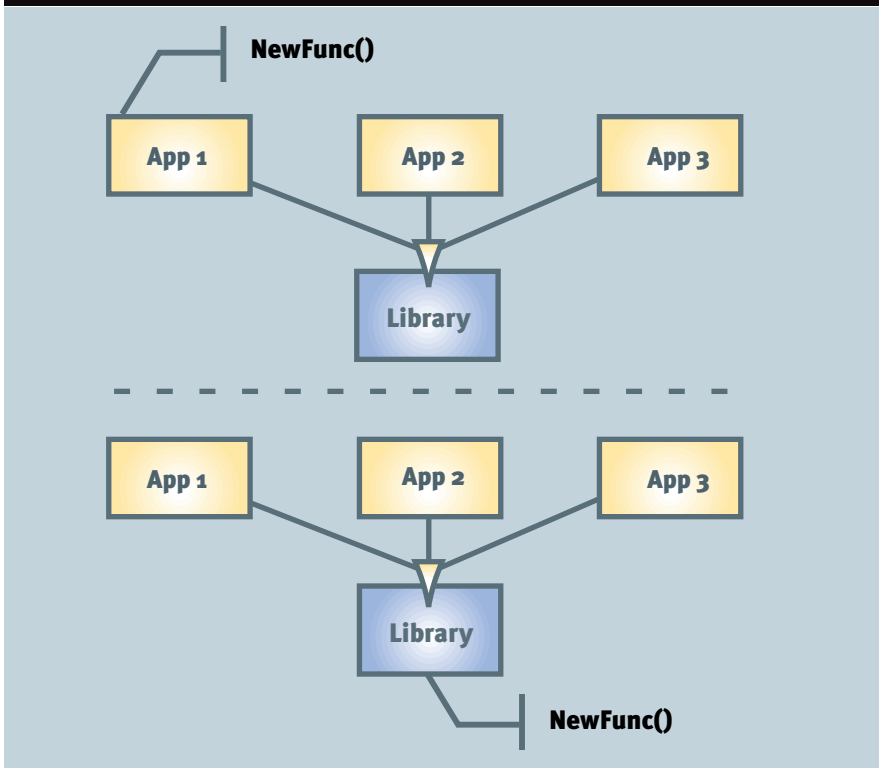
An advantage of the virtual manager system is that most new code can first be introduced at the application level without modifying the existing libraries. If you want to use those features later in other applications sharing the same libraries, then it's simple to migrate the code from the application level to the library level (Figure 4). It's important to note that when `NewFunc()`, as shown in the diagram, moved from App 1 to the Library, no changes to the interface were apparent to the rest of the program. The beauty of this system is that migrating code in a virtual manager class doesn't affect the interface as seen by the rest of the program.

In addition to the ease with which you can migrate code from application to library, the virtual manager also creates, to a limited extent, a primitive

versioning scheme. You can see how, by creating several layers of derived classes, you could easily choose at the application level which version of a library you wished to use.

While there certainly is no magic formula to eliminate bugs from a software product, a well-planned and solidly-developed library can certainly go a long way in stabilizing your code. By utilizing well-established object-oriented programming principles and solid code management techniques, you'll find that perhaps you'll be able to focus more energy and spend more time writing new code instead of tracking down bugs and rewriting old libraries. ■

FIGURE 4. *Migration of code from application-level facade to library-level facade.*



FOR FURTHER INFO

Gamma, Erich; Helm, Richard; Johnson, Ralph; and Vlissides, John
Design Patterns: Elements of Reusable Object-Oriented Software
(Addison-Wesley, 1995)

POSTMORTEM

Aiii
HEAVY GEAR II

by Clancy Imislund

After a long series of successful titles such as MECHWARRIOR 2: GHOST BEAR'S LEGACY and MECHWARRIOR 2: MERCENARIES, Activision held a dominant position in the giant-robot genre. Due to the commercial success of this series, though, a tidal wave of similar products developed by worthy competitors began to flood the market. Fortunately, Activision found a new and exciting universe in Dream Pod 9's Heavy Gear pen-and-paper-based game, and in the fall of 1997, Activision Studios began production on the futuristic giant-robot simulator HEAVY GEAR II. HEAVY GEAR II allows game players to suit up in a giant, high-octane, humanesque battle tank called a "Gear." The player is then required to outfit a wily band of AI Gears (called "squadmates") and arm them to the teeth. Their small but

Beyond the role of volcanic guitarist and avid jai alai analyst, Clancy enjoys all aspects of engine design including 3D graphics, AI, tools, and multiplayer integration. For HEAVY GEAR II, he was responsible for the AI, scripting system, and tools development. Contact him at cimislund@activision.com.

heavily armed reconnaissance force must infiltrate and overcome a rich variety of environments including swamps, icy wastes, angry red planets, and the weightless reaches of outer space. The action in HEAVY GEAR II is much faster than in other giant robot simulators, as Gears are substantially smaller than Mechs and much more fragile. This requires squads to rely primarily on stealth and cunning rather than brute force and wanton destruction. Players are forced to pick their battles wisely or face annihilation from the superior enemy legions that infest the various worlds they traverse. When pilots are tired of playing against computer opponents, they can pit themselves against human enemies in multiplayer mode with a number of game types to choose from including "steal the beacon," "king of the hill," and good old-fashioned deathmatch.

To create such an experience, Activision assembled a new team of its best and brightest in the areas of programming, art, design, and management. A very aggressive schedule was adopted targeting a fall 1998 release date (a 13-month development schedule). Our original staff consisted of a lead, graphics, AI and tools, multiplayer and shell programmer. We had a lead artist, two 3D modelers, a 3D animator, and two 2D artists creating texture maps for the models and terrain. We also had a lead designer who managed a group of three game designers, all of whom transformed the storyline from paper into a computer game. Our management group consisted of a director, a producer, and an associate producer, who controlled the schedule and the development and direction of the game and saw to the many needs of the other production staff.

We programmers were chartered to create a new game engine and I took on the role of AI, scripting system, and tools programmer. At the heart of this engine was a rock-solid memory management and leak-tracking class. Yes, that's right. Before anyone ever dreamed of fancy graphics or stunning game play, we had to deal with this mundane task. Every C++ class and structure used by the Darkside engine had roots within this base class. This architecture allowed to us to detect memory leaks and overwrites as soon as they appeared in a given day's build, which allowed us to address problems immediately rather than during a grueling cycle at the end of the project. I cannot stress the importance of this type of planning and execution enough for teams who want to craft a state-of-the-art game engine. Focusing on the reliability of the application will also greatly increase the immersion factor of any game that's created with it. After all, what destroys immersion more than a hard system crash? Hats off to our lead who took us down this path.

The decision was made to target the game only for machines outfitted with 3D-accelerated video cards. This issue was hotly disputed within our team for a while, but when our management group realized that in order to pull off realistic 3D graphics, software emulation was a dead deal. This had profound effects on the schedule, as it freed our artists and programmers from dealing with the time-consuming and tiresome work of generating alternate LODs for such a purpose. This decision also eliminated a huge chunk of our QA test plan, which could have pushed back the release date of the title significantly further. We had to choose between putting out the highest quality game we could and targeting a consumer market that barely existed at that point, or publishing a game that had a greater current market base and little or no novelty by the time it was released. I suppose it is something all of us developers must deal with in these days of rampant technological change.

Artist's concept for the "Asteroid Shipyards."

The AI system was another great challenge, as it would dictate much of the feel of the game. It would also be a key tool for our design team as they implemented the complex storyline. We decided at the high level to go with a pure "autopilot" approach, in which enemy and friendly AI alike were able to function intelligently without any script at all for individual units. We wanted to be able to put a unit into the world and have it go do whatever it thought it needed to do. Internal to this was a high-level strategic system, a team-based tactical system that employed a knowledge base, and a low-level, unit-specific order execution system. The goal was to take much of the burden off of the design staff, and give the game a consistent and distinctive feel across all missions.

Nonetheless, our scripting system was very effective and had some nice features such as an in-game debugger complete with break points, single-step execution, and variable watch windows. It was based on LEXX/YACC-based C grammar, which hooked into a powerful and easily extensible virtual machine that resided in the Darkside engine. Our scripting system was hooked in to our custom game API, which in turn was coupled to prototypes defined in a file called STDHG2.H, which ostensibly replaces STDIO.H (a file well known to all of you C programmers out there). Amazingly enough, STDHG2.H was used not only to compile the

HEAVY GEAR II

Activision
Santa Monica, Calif.
(310) 255-2000
http://www.activision.com
Release date: July 1999
Intended platform: Windows 95/98
Project length: 19 months
Team size: 20
Critical development hardware: 3D-accelerated 200MHz PC
Critical development software: Softimage 3.7, Photoshop, Visual C++ 5.0, Visual SourceSafe, and numerous in-house tools

Mining installation on the wastes of Caprice.

Early concept for the destruction at "Peace River."

scripts, it was also used in the compilation of the Darkside engine itself. This convenient relationship between the scripting language and the engine source code, plus the fact that the C language is widely documented, easily justified our decision use a scripting language based on C. Scripts were used mainly to monitor mission progress and objectives, special behaviors (convoys and patrols, for instance), and interactive control of the action. Our goal was to keep these scripts as simple as possible. The autopilot handled all strategic, tactical, and low-level operations of unit behavior, yielding control only at the request of a script. The autopilot was the default AI handler in the engine. Scripts could override this system by posting event callback requests as shown in Listing 1.

LISTING 1. *Event callback requests.*

```
// A Script.C
void
Initialized()
{
   // Tell AutoPilot that we want control
   // back when the unit is killed or
   // shot at
   AutoBreak(HitPointsExhausted);
   AutoBreak(ShotAt);

   // Turn over control to the autopilot
   // when this function returns
   AutoPilot();

}

void
ShotAt()
{
   RegisterString("Ha ha, ya missed");
   // Resume auto mode
   AutoPilot();
}
void
HitPointsExhausted()
{
  RegisterString("Damn! I died!");
}
```

Our multiplayer system was crafted using a proven proprietary networking SDK developed at Activision. This reliance on preexisting technology allowed us to get multiplayer functionality in the engine and working very early on in the development cycle. Designing and integrating multiplayer functionality is often left until the end of a project, and that can create all manner of problems. Getting this level of complexity into our development schedule early probably saved the HEAVY GEAR II team an additional six months of work.

What Went Right

1. EFFECTIVE PROTOTYPES. When I began working with Activision's HEAVY GEAR II production team, it was composed of a lead programmer, a top-of-the-line 3D graphics programmer, a director, a producer, a lead designer, and a lead artist. At that point, the team had just been given marching orders to produce a second prototype of the game for approval by the corporate brass. I was hired because this prototype required functional AI to give a feel of the intended game play. At this early stage, the game already looked super and a user-friendly 3D layout tool for the level designers was up and running. I couldn't believe the amount of work that these people put into producing the first prototype. Even more amazing was the fact that most, if not all, of this work was of the "keeper" variety — nearly all the code used in the green-light prototypes exists in the Darkside engine today. Our design tools were augmented in functionality, but the basic underlying technology driving these applications remained virtually unchanged through the entire development cycle.

2. THE DARKSIDE ENGINE. The Darkside engine created specifically for HEAVY GEAR II was an engineering gem. Although game players get a solid dose of its extraordinary graphics and animation capabilities, at its core it is nothing but a simple memory manager and leak tracker. This property of the engine allowed the programmers to track down and remedy most of the nastier bugs in the game long before it hit the QA floor. Beyond its concrete foundation, its modular design and expert use of C++ made it a snap to add or delete the different systems we wished to experiment with. Most of the foundation code and graphics subsystem was shared with our 3D layout tool, saving us a ton of extra work. This extensibility and proper use of the C++ language was invaluable as we faced a deluge of changing design requirements and additional game features. Our 3D math library was easily modified for Intel's Katmai Instruction Set, and this enabled our OEM group to strike up some valuable partnerships with external vendors. Although some HEAVY GEAR II–specific code may still lurk in the recesses of the Darkside, the engine's debug ability and modularity make it a novel choice

for any 3D simulator Activision may wish to develop in the future.

3. They let us do our jobs. Those of you who have had the experience of a tough supervisor or management group breathing down your neck as you tiptoed down your critical path will appreciate this: our front-line governing body of director and producer did a superb job of trusting the professionals who worked under them. They made our team aware of upcoming milestones and the expectations of Activision's corporate group without succumbing to the temptations of micro-management and general megalomania. This held true even when the team hit roadblocks or missed milestones. They maintained their trust in us. This contributed to a fertile environment for new ideas and creative solutions from the team.

Activision's upper-level management also contributed to the game's success by letting the team decide when the game was finished. Management could have released the game early and announced a patch shortly thereafter, a practice that is becoming common with many publishers nowadays. Instead, Activision gave game players a break and released a quality, bug-scarce title with a lot of replayability and immersion. This is an admirable goal in this era of market-driven development and patch-laden gaming web sites.

4. Great QA work. With Activision's business plan, its development teams enjoy the luxury of a highly organized QA department. On the front lines we have a smaller group known as "production testers." These folks deal directly with the development team on a daily basis and intercept a high percentage of the bugs before they ever make it out to external test groups and Activision's QA team proper. Production testers become very intimate with the inner workings of the game and are actually solicited for new ideas to improve the design. When the development team and production test team feel that the title has reached beta, Activision's main QA test group takes over for a formalized certification process. Concise reports are painstakingly generated to ensure that only genuine, replicable bugs ever make it back to the development teams. It is this group that is ultimately responsible for the release of the title, so they take their work very seriously.

Beyond Activision's in-house QA department, we were aided by an eager external group called "Visioneers" who avidly played any build we furnished for them. They reported all manner of problems, from hardware compatibility issues to boring game play. The size and diversity of this group made them an ideal QA avenue, as they represented a more accurate cross-section of the gaming community. Activision also invited people from different age groups to come into our office and play HEAVY GEAR II in an observational setting. Testers' responses were noted and given to the development team via written surveys. All these channels of software evaluation greatly streamlined the development process and contributed heavily to the quality of the finished product.

5. Good game pacing. If you sit down and play HEAVY GEAR II, you will notice a completely different feel from other games in the genre. The pace is faster and the action is much more compelling. This was a desired feature documented in the original specification of the game, but writing down that requirement in a document doesn't necessarily mean you will pull it off. At about the same time that HEAVY GEAR II was kicking off its production cycle, a series of first-person action shooters hit the streets. Most if not all of the team members were avid deathmatch players, and fierce competitions were held to decide which of us was the top dog. Subconsciously, much of the gameplay feel we experienced while partaking of these frag-fests found its way into our title. A fun enemy AI unit was one that somehow evoked the same emotional response as the poor slob I had just fragged at lunchtime.

This held true for the HEAVY GEAR II multiplayer experience as well. We tried to translate what we thought was fun and exciting in our lunchtime deathmatches into experience of fighting against hulking Gears. I'm not saying that if you want to create a great title with lots of excitement that you should play first-person shooters, but I do think that it is important for all game developers to be avid players as well. It is the game player that has the upper hand at identifying the abstract notion of "fun" and, isn't it the job of the developer to create that?

Early sketches of CEF Heavy Battle Frame.

What Went Wrong

1. To demo or not to demo. I love to play the demo version of any game before I buy it. In fact, as a gamer I purchase games based on how good

the demo is. As a developer, however, a demo version of a game is a tricky thing to pull off. In our case, most of the game's critical systems were not functioning on all cylinders and many game assets didn't exist yet. We had to dedicate most of our time and resources to this Cimmerian task, although all our schedule called for was a small tiger-team consisting of a designer, programmer, and artist who worked on it a small percentage of their day for about two weeks.

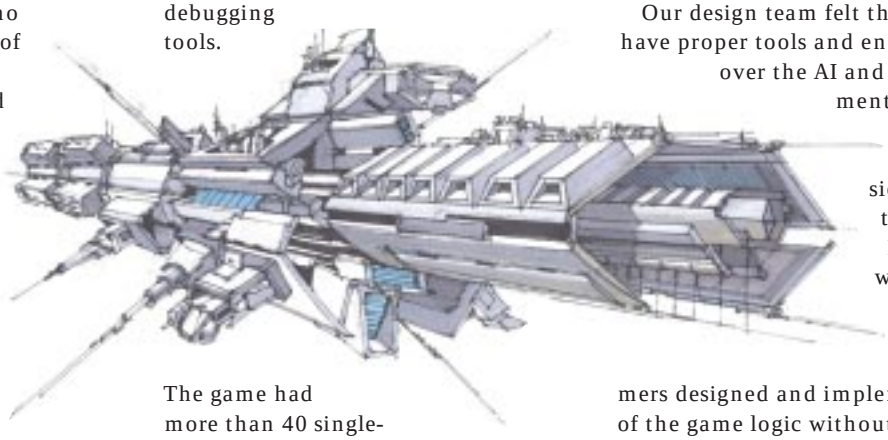
When the smoke cleared and the demo was finally posted, the team gave a sigh of relief and basked in the warmth of a job well done. Unfortunately, this feeling quickly evaporated when we realized that we had taken almost a three-month departure from our original development schedule and totally exhausted ourselves in the process. I still believe a demo is important to the success of a game, but such a task should be closely correlated to the production of the main SKU. Attempting to factor the demo development time and resources into a schedule significantly different from the game itself can profoundly affect both the product's quality and timeliness.

2. UNDOCUMENTED TURNOVER. This is a very common yet unpredictable aspect of game development. New and better jobs or personal issues always seem to snatch away even the most loyal and dedicated teammates. As a result, someone inherits the workload of the fallen comrade, putting the team and the schedule in a precarious situation.

When turnover occurred on the HEAVY GEAR II team, we and our schedule encountered a nasty surprise. Many of the systems we inherited were only partially implemented and virtually undocumented. To make things worse, much of the code was written to implement advanced animation and mathematical methods used all over the game engine, and we didn't have a technical design document that we could refer to in order to determine the intended solution for these systems. Working around the bugs and limitations in these systems cost the team even more time and generated bushels of frustration. This aspect of the turnover phenomenon is the least respected by developers and has the most profound and variable effect on scheduling. A periodic and thorough code review process is an effective way to defeat this problem.

3. SCHEDULE TOO AGGRESSIVE. HEAVY GEAR II was built completely from the battleground up, so every aspect of the game required significant development time. Unfortunately, the development schedule was too optimistic about how long it would take to create the title.

Here's an idea of the scope of the game, as specified by our design document. HEAVY GEAR II required a brand new game engine and an accompanying suite of design, development, and debugging tools.



The game had more than 40 single-player missions, as well as numerous historical and instant-action missions. Additionally, multiplayer functionality (including cooperative play with multiplayer AI) was required for deathmatch, king of the hill, steal the beacon, and historical settings. We were also required to construct three different modes for each and all of these missions: terrestrial, space, and "Gomorrah" (combat in an enclosed, multi-level, near-future megacity) gaming modes.

The schedule called for a polished demo version of the game to be posted on all of the top gaming web sites. I joined Activision on November 10, 1997, after HEAVY GEAR II's first prototype, and the final product was initially slated for release for Christmas of the following year. This period included a protracted QA run-through, effectively yielding a nine-month development cycle. Unfortunately, it was far too aggressive. Although we implemented all of the required features as defined in the original design specification (and many not included), we missed our final ship date of October 1998 by nine months.

4. ELEVENTH-HOUR SOLUTIONS. During the production of HEAVY GEAR II, many of our game designers and external testers began to notice a distinct absence of thrilling game play, which were attributed to several factors: the AI was too vicious, the AI-controlled squadmates were disobedient and difficult to deal with, and there was no noticeable ramp-up in difficulty and emotion — some missions were frightfully easy and others absolutely impossible to complete. (The average life expectancy of the player in some of our combat scenarios was about four seconds.)

Our design team felt that they didn't have proper tools and enough control over the AI and the environment to adequately tune their missions and make the game fun. Most of this was due to the fact that we arrogant programmers designed and implemented most of the game logic without enough consultation and input from our able game design staff. To address these issues, our team had to depart totally from the design document and do a whole bunch of wild and creative thinking. Only after some ad hoc brainstorming sessions and grueling mission-by-mission playability tests did the pieces come together. Much of the kudos we received from the gaming community is directly related to these solutions.

However, a situation like this could have had far worse results. The next time around we will put much more thought and detail into our game design documents and provide a much more efficient education for our game design staff concerning game play manipulation and control via scripts. We will also give them a more prominent role in the initial design of these systems to enhance their understanding of the underlying technology.

5. Marketing issues. After experiencing the high quality of our second prototype, people at Activision began to get very excited about the future of HEAVY GEAR II. We were given a highly favorable reception at E3 in 1998, where we finally revealed the game. We subsequently released a playable demo that met with similar acceptance. Our marketing staff, which had been pushing this title from the start, finally had the leverage it needed to differentiate HEAVY GEAR II from its competitors. Magazine articles began to run, OEM deals were made, and shelf displays were created as everyone anticipated the forthcoming release.

Conceptual art for gateship "Celestrus" orbiting barren Caprice.

Only the development team knew how alarmingly behind schedule the game actually was, however. This was due in part because of some unfortunate turnover in our staff, but the main factor was the amount of time and effort we had allocated to creating the demo. We had hit roadblocks in the past and always rebounded in sterling fashion, so nobody on our team allowed themselves to believe that the demo would cause us to miss our target ship date.

Then we slipped. Suddenly we were plunged into a nearly interminable crunch mode. Our slip meant that the game wouldn't ship in time for the Christmas season, so we decided to shoot for a more realistic March release. Our marketing group did their best to respond to the blow and attempted to keep interest in HEAVY GEAR II alive. March came and went, and the release date became a dancing phantom beyond our reach. We developers knew we were close to completing the game, but nobody could give our marketing team a definite date so that they could keep the buzz up. Marketing did what it could to keep whetting the public's appetite as the weeks rolled by.

Missing the targeted ship date is a serious risk to teams that rely on new and unproven technologies — and it's especially perilous for teams working under compressed development schedules. Even the most innocuous development tasks, if underestimated or mishandled, can send your schedule flitting away beyond your control.

We Almost Got It Right...

We received great reviews from top gaming publications and web sites. Our marketing group piqued the interest of the gaming community and our development team produced a superior title. And since our development team had the luxury of starting this project from scratch, we had the opportunity to learn a wide range of new methods and techniques that would otherwise have remained beyond our reach.

Alas, our aggressive schedule and risk taking proved to be our Achilles' heel. Shipping a quality title is important, but so is strict adherence to budget and schedule. The HEAVY GEAR II team learned a great deal from this experience; we'll keep that with us for a long time. At least we left a solid and reusable game engine in our wake. ■

ADVERTISER INDEX

U.S. Postal Service Statement Of Ownership, Management And Circulation (Required by 39 U.S.C. 3685) (1.) Publication Title: Game Developer (2.) Publication No.: 1073-922X (3.) Date of Filing 1-Oct-99 (4.) Issue Frequency: Monthly (5.) No. of Issues Published Annually: 12 (6.) Annual Subscription Price: $49.95 (7.) Complete Mailing Address of Known Office of Publication: Miller Freeman Inc., 600 Harrison Street, San Francisco, CA 94107 (8.) Complete Mailing Address of the Headquarters of General Business Office of the Publisher: Miller Freeman Inc., 600 Harrison Street, San Francisco, CA 94107 (9.) Full Names and Complete Mailing Addresses of Publisher, Editor, and Managing Editor. Publisher: Cynthia Blair, Miller Freeman Inc., 600 Harrison Street, San Francisco, CA 94107. Editor: Alex Dunne, Miller Freeman Inc., 600 Harrison Street, San Francisco, CA 94107. Managing Editor: Kimberley Van Hooser, Miller Freeman Inc., 600 Harrison Street, San Francisco, CA 94107 (10.) Owner: Miller Freeman Inc., 600 Harrison Street, San Francisco, CA 94107, a wholly owned subsidiary of United News & Media plc, Ludgate House, 245 Blackfriars Road, London SE1 9UY, England (11.) There are no Known Bondholders, Mortgages, or Other Security Holders Owning or Holding 1 percent or More of Total Amount of Bonds, Mortgages, or Other Securities (12.) Does not apply (13.) Publication Name: Game Developer. (14.) Issue Date for Circulation Data Below: October 1999. (15.) Extent and Nature of Circulation / Average No. Copies Each Issue During Preceding 12 Months: A. Total No. Copies (Net Press Run): 44,947 B. Paid and/or Requested Circulation (1.) Sales Through Dealers and Carriers, Street Vendors, and Counter Sales: 2,732 (2.) Paid or Requested Mail Subscriptions: 31,553 C. Total Paid and/or Requested Circulation (Sum of 15B1 and 15B2): 34,284 D. Free Distribution by Mail (Samples, Complimentary, and Other Free): 1,439 E. Free Distribution Outside the Mail (Carriers and Other Means): 1,283 F. Total Free Distribution (Sum of 15C and 15F): 37,007 H. Copies Not Distributed (1.) Office Use, Leftovers, Spoiled: 1,555 (2.) Return from News Agents: 6,385 I. TOTAL (Sum of 15G, 15H(1) and 15H(2): 44,948. Percent Paid and/or Requested Circulation: 92.64%. Actual No. Copies of Single Issue Published Nearest to Filing Date. A. Total No. Copies (Net Press Run): 43,366 B. Paid and/or Requested Circulation (1.) Sales Through Dealers and Carriers, Street Vendors, and Counter Sales: 2,460. Paid or Requested Mail Subscriptions: 32,167 C. Total Paid and/or Requested Circulation (Sum of 15B1 and 15B2: 34,627 D. Free Distribution by Mail (Samples, Complimentary, and Other Free): 1,569 E. Free Distribution Outside the Mail (Carriers and Other Means): 0 F. Total Free Distribution (Sum of 15d and 15e): 1,569 G. Total Distribution (Sum of 15c and 15f): 36,196 H. Copies Not Distributed (1.) Office Use, Leftovers, Spoiled: 1,695 (2.) Return from News Agents: 5,475 I. TOTAL (Sum of 15g, 15h(1) and 15h(2): 43,366. Percent Paid and/or Requested Circulation: 95.67%. I certify that the statements made by me above are true and complete (signed) Kimberley Van Hooser, Managing Editor.

SOAPBOX

by Warren Spector

Go, Team!

Human nature being what it is, I was hugely flattered to be asked to participate in the September 1999 *PC Gamer* article about the 25 most talented people in gaming — the one they titled "Game Gods," leading to no end of entirely justified ribbing around Ion Storm's Austin offices....

Ribbing aside, such an honor represents an almost unmatchable expression of respect from journalists, peers and gamers — the sort of thing one works a lifetime to achieve. It may surprise you, then, that I almost turned the opportunity down.

Why?

Well, the crux of the biscuit is that it seems unseemly and, more important, inappropriate to single people out for "star treatment" in a business as intensely collaborative as game development. Before getting into gaming, I always figured I'd end up making movies and spent a lot of time studying that business — you know, "film, the collaborative art...." Well, I'm here to tell you that there is no more collaborative medium than gaming. The movies got nothin' on us, friends. There are so few renaissance game creators it's hardly worth the effort of identifying and listing them.

In fact, honoring individuals represents an almost criminal denial of the critical contributions of the dozens of team members who are, if anything, more responsible for the success of the games you know and love than the individuals typically credited with the creation of those games. And the elevation of individuals to star status, while understandable in this increasingly marketing-driven age, symbolizes much of what I dislike about the game business as we approach the millennium. But let me be more specific.

I've been credited with the creation of UNDERWORLD and SYSTEM SHOCK, honored as "The Man" behind ULTIMA VII, PART 2: SERPENT ISLE, cited as the creative force behind the "underappreciated" WINGS OF GLORY.

illustration by Jackie Urbanovic

As anyone who knows me will tell you, I'm intensely proud of those titles and my contributions to them. All of them appear on my résumé, points of pride and high-water marks in a career that also includes some real clinkers. (Thankfully, no one much talks about the bad ones anymore but buy me a drink sometime, and I'll tell you horror stories....) Frankly, I find being credited with those titles — the good and the bad — vaguely embarrassing. It would certainly be the height of arrogance for me to take sole or even majority credit for them.

UNDERWORLD and SYSTEM SHOCK would have amounted to nothing — wouldn't have happened at all — without the initial impetus provided by Blue Sky Productions' founder Paul Neurath. UW and SHOCK would never have been as cool or memorable as they were without the inspirational leadership of project director Doug Church (the Most Talented Individual I've worked with in this team-oriented business, if you must know). And let's not forget the contributions of programmers like Marc LeBlanc, Rob Fermier, Art Min, Jon Maiara, Dan Schmidt, and James Fleming, or designers like Tim Stellmach and Dorian Hart, or audio guys like Greg LoPiccolo and Eric Brosius, or testers like Harvey Smith. UNDERWORLD and SYSTEM SHOCK are their creations, not mine. More accurately, they are our creations, all of us applying our unique, individual talents to the accomplishment of mutually agreed-upon goals.

Similarly, SERPENT ISLE is the product of more than 30 hearts, minds, and souls. I came up with a tone and "feel" I wanted to achieve and a story concept. I set some parameters on the world and characters. But the minute-to-minute details of the storyline were fleshed out by some amazingly talented designers — Steve Powers, Dave Beyer, Bill Armintrout, and others. And I watched, usually with jaw on ground, as lead artist Denis Loubet and the rest of the art team brought to life the world and characters of SERPENT ISLE. And without testers like Marshall Andrews, the game wouldn't have been half what it ended up being. I built not one inch of the map, wrote not one line of dialogue, implemented not one game function. So whose game is it?

Continued on page 71.

Warren Spector runs Ion Storm's Austin, Texas, office. He is currently working on a new role-playing game, DEUS EX. In the past, he has produced games for Origin and Looking Glass Technologies. You can reach him at wspector@ionstorm.com.

Continued from page 72.

WINGS OF GLORY was largely the same thing. I came up with an idea, a feeling I wanted to evoke during play. I worked with a design team ultimately led by Dave Beyer to craft a story and basically nodded my head a lot in stunned agreement as lead artist Whitney Ayres crafted the perfect look for the game. I watched, often dumbstruck, as programmers Bill Baldwin, Tony Bratton, and John Talley brought to near perfect life the vision I had in my head when we started. Who deserves credit for WINGS OF GLORY?

When "Warren Spector's DEUS EX" comes out, will people acknowledge the incredible contributions of lead programmer Chris Norden, lead designer Harvey Smith, lead artist Jay Lee, and the rest of the team? Odds are, they won't and I'll end up writing more articles and posting more Usenet messages and talking to more journalists to convince gamers that individuals don't make great games — great teams do.

Now, maybe there are a couple of guys in the list of "Game Gods" who are one-man shows. I certainly find myself in awe of John Carmack, Peter Molyneux, Sid Meier, and some others in the *PC Gamer* group. (And, man, did I geek out and have a great time hanging with them!) Maybe I'm the anomaly. I doubt it, though. I'm willing to bet we're all team-oriented guys.

I suspect the impetus to single out one person for credit (or blame) is some kind of human need for shorthand, a way to separate wheat (good games) from chaff (bad games) with some simple formula based on past successes. Or maybe the spotlight turned on individual game developers, making some of us "brand names," is just an easy way for marketing guys to earn their salaries. I don't know.

What I do know is that I'd be nowhere without the teams that have backed me up and often dragged me, kicking and screaming, toward success. And before anyone starts assuming I'm being falsely modest, let me assure you I have quite enough ego for several ordinary mortals. I know I'm a good designer, a good process manager, a good people manager, a good business guy, and I'm even O.K. at the PR end of things. There are certainly others who are better than I am at any one of those things, but put them all together and you've got a package I've come to see as pretty rare over the years, one that allows me to play to the strengths and help overcome the weaknesses of almost any team. I'm not being modest. But "Game God"? Hmm?

Anyway, if you're a producer, project director, or Game God, reflect for a moment as you talk to the press, publishers, and PR people on all the unsung heroes whose names are never mentioned. Think of their contributions to "your" success and to the "individual" successes of the other lucky folks lauded on gaming web sites and in the pages of game magazines. Give credit where credit is due. I know it's hard. I know the press doesn't want to hear this. But do it anyway because it's the right thing to do.

And if you're one of the unsung, well, all I can say is hang in there. We may live in a PR world nowadays, but your contributions are appreciated by people who "get it." And maybe — just maybe — if we all start talking about this, someone will start listening and you'll get your well-earned day in the sun. ■