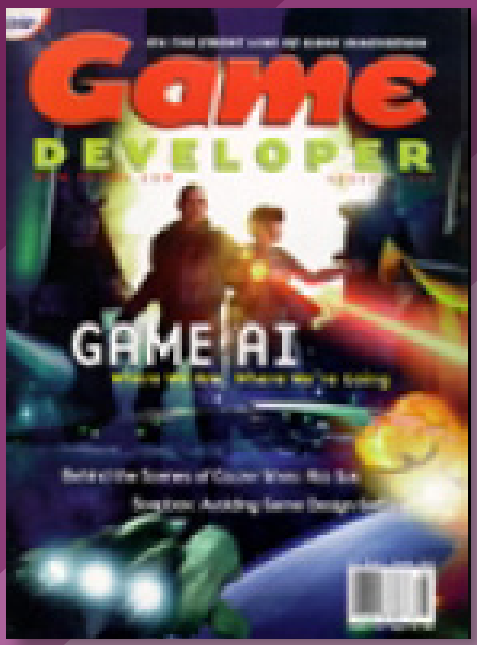
gd

GAME DEVELOPER MAGAZINE

AUGUST 2000



UBM Tech

Return

GAME PLAN

LETTER FROM THE EDITOR

Fun & Games With Cell Phones

At E3, it's typical to gravitate toward the glitzy titles on display. But in a relatively subdued booth in the lobby of the Los Angeles Convention Center, I found Nokia quietly telling people about their plans to bring more games to wireless devices.

Nokia has huge plans for cell phones and games. The company estimates that by 2002 more than a billion people will have mobile phones, and by 2003 more people will access the Internet through wireless devices than by PCs. Those are some huge numbers, and it puts the companies who control cellular networks in the catbird seat. Nokia, for one, plans to make games a central application on cell phones. Of course, access to a large installed base of customers doesn't ensure the viability of a platform. (I think the slow progress of interactive television indicates what a well-coordinated effort it takes to bring a new interactive platform to market and make it succeed.) But Nokia is quietly moving forward into the games market, and I like the company's vision for the future.

With the launch of the Nokia Mobile Entertainment service (a one-stop shop that will let mobile network operators serve games to their mobile subscribers), the company is positioning itself as a provider of an online server platform in the world of wireless gaming devices. In theory, someday your cell phone will have an "always on" GPRS (General Packet Radio Service) connection to your network operator, and when you want to play a game, you will select a game from among those currently offered and begin playing. There will be no numbers to dial and billing will be automatic.

As a developer, you would approach a Nokia Mobile Entertainment game publisher to get your game up on the service. (If you're a publisher, you would contact Nokia directly.) In this revenue model, the network operators would take their cut, Nokia would take its cut, and you the game developer would get your share, too, all based upon the revenue the game generated from users. Whether consumers would pay a flat fee or some kind of hourly charge based on usage would probably be up to the network operators.

Of course, there are hurdles to contend with. The most graphically impressive demo that I saw on the Nokia booth was a 2D chess game, and the whole chessboard couldn't fit onto the small cell phone display. So improving the graphics capability of cell phones is critical. One tool that might help in this regard is n3D, a graphics API that 3D Pipeline of La Jolla, Calif., is creating for use in cell phones, PDAs, and set-top boxes. According to Greg Passmore of 3D Pipeline, n3D is very modular and fits in a tight memory space. It will include a sample game engine, a 3D API, a 2D API, support for multiple colored lights, anti-aliasing, and a scene graphics manager that handles BSP culling, progressive transmission, collision detection, and more.

Nokia must also work with other companies to make the Wireless Application Protocol (WAP) the standard way for mobile phones in the U.S. to communicate with servers installed in the mobile phone networks. Unlike Europe (which has embraced WAP), there is no dominant protocol for this in the U.S., and until that changes, support for any given wireless application will be spotty and market growth will suffer.

If and when games for cell phones take off, small (even single-developer) game development companies could benefit. Because of their limited capabilities as gaming devices, cell phone games themselves will have to be fairly simple (the potential of n3D and its implications for the platform notwithstanding), and as such solo game developers might face a more level playing field against deep-pocketed game companies.

For more information about Nokia's plans, check out www.nokia.com/wap/entertainment.html.

Let us know what you think. Send e-mail to editors@gdmag.com, or write to *Game Developer*, 600 Harrison St., San Francisco, CA 94107

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
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL

Editorial Director
Alex Dunne *adunne@sirius.com*
Managing Editor
Kimberley Van Hooser *kvanhoos@sirius.com*
Departments Editor
Jennifer Olsen *jolsen@sirius.com*
News & Reviews Editor
Daniel Huebner *dan@gamasutra.com*
Art Director
Laura Pool *lpool@cmp.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Jeff Lander *jeffl@darwin3d.com*
Lisa Washburn *article@vectorg.com*
Advisory Board
Hal Barwood LucasArts
Noah Falstein The Inspiracy
Brian Hook Verant Interactive
Susan Lee-Merrow Lucas Learning
Mark Miller Group Process Consulting
Paul Steed
Dan Teven Teven Consulting
Rob Wyatt Microsoft

ADVERTISING SALES

National Sales Manager
Jennifer Orvik e: *jorvik@cmp.com* t: 415.905.2156
Account Executive, Western Region & Asia
Mike Colligan e: *mcolligan@cmp.com* t: 415.356.3486
Account Executive, Northern California
Susan Kirby e: *skirby@cmp.com* t: 415.356.3406
Account Executive, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 415.905.2323
Sales Associate/Recruitment
Morgan Browning e: *mbrowning@cmp.com* t: 415.905.2788

ADVERTISING PRODUCTION

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

CMP GAME MEDIA GROUP MARKETING

Marketing Manager Susan McDonald
Product Marketing Manager Darrielle Sadle
Field Marketing Manager Jennifer McLean
Marketing Coordinator Scott Lyon

CIRCULATION

BPA INTERNATIONAL
Game Developer magazine is BPA approved

Vice President/Circulation Jerry M. Okabe
Assistant Circulation Director Kathy Henry
Circulation Manager Stephanie Blake
Circulation Assistant Kausha Jackson-Crain
Newsstand Analyst Pam Santoro

INTERNATIONAL LICENSING INFORMATION

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

CORPORATE

President & CEO Gary Marshall
COO/Corp. President, Business Tech & Channel John Russell
President, Business Technology Group Adam Marder
President, Specialized Technology Group Regina Ridley
President, Channel Group Pam Watkins
President, Electronics Group Steve Weitzner
General Counsel Sandra L. Grayson
Vice President, Creative Technologies Johanna Kleppe
General Manager, CMP Game Media Group Greg Kerwin

CMP

FRONT LINE TOOLS

WHAT'S NEW IN THE WORLD OF GAME DEVELOPMENT | daniel huebner

PLAYSTATION GETS SMALL

Sony is entering the portable game system fray with a remarkably cute miniature version of the venerable 32-bit Playstation. The new machine, cleverly called PS One, comes in at just one-third the size of the original system but with all the original functionality. In addition, the PS One will add network connectivity in the form of a cell phone connection capability. As a challenger to Nintendo's Game Boy Color and Game Boy Advance, however, the PS One faces some serious limitations; the machine won't run on batteries and will need to be plugged into an AC outlet or car lighter. Also, a four-inch LCD screen will be sold as an optional accessory rather than being included with the machine. The PS One will land in North America this September.

PS ONE | Sony | www.sony.com

VOODOO 5 ARRIVES

3dfx has finally pushed its Voodoo 5 out the door after a brief delay and recall to iron out some last-minute problems. Offering application-independent real-time full-scene hardware antialiasing compatible with all major APIs, the Voodoo 5 5500 AGP is based on dual 3dfx VSA-100 processors with the T-Buffer digital cinematic effects engine and 64MB of total graphics memory. 3dfx is pricing the Voodoo 5 5500 AGP at an aggressive $299.99.

VOODOO 5 | 3dfx | www.3dfx.com

REALVIZ ANNOUNCES RETIMER 2

Realviz is releasing a new version of its image sequencing timer, ReTimer. ReTimer 2, which allows users to speed up, slow down, or adjust the timing of an image sequence, offers several new features, including a new engine. Whereas version 1 used a vector grid, each pixel now has its own computed motion (dense flow) where the user can define the density of the motion field displayed for better control over output quality. ReTimer 2 is available for Windows NT, IRIX, and Linux, and is priced at $3,500, which includes one ReTimer license, one batch license, and three months of free maintenance (support plus upgrades). One year of maintenance costs an additional $525.

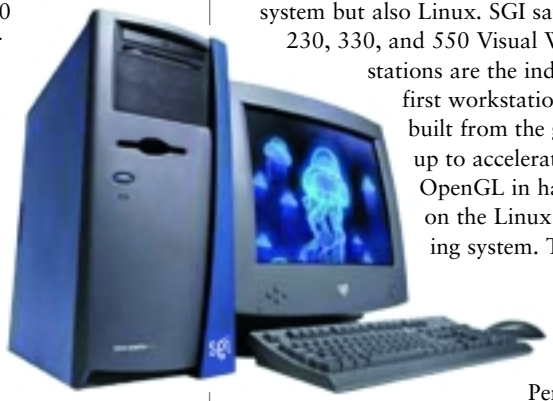
RETIMER 2 | Realviz | www.realviz.com

RAD TOOLS FOR MAC

RAD Game Tools is doing its part to make the Macintosh a more friendly place for game development. The company started its new Mac crusade with MacOS versions of its immensely popular Bink and Smacker 4 video codecs, and is now widening its offerings by bringing the Miles Sound System and Granny 3D animation system to the Macintosh as well. For its part, Alias|Wavefront is also jumping on the Mac bandwagon and has a MacOS version of Maya in the works.

RAD GAME TOOLS | www.radgametools.com

SGI ROLLS OUT NEW VISUAL WORKSTATIONS



SGI is introducing a new line of Intel processor-based workstations, offering not only the Windows NT operating system but also Linux. SGI says its 230, 330, and 550 Visual Workstations are the industry's first workstations ever built from the ground up to accelerate OpenGL in hardware on the Linux operating system. They include either single or dual Pentium III or Pentium III Xeon processors, a 133MHz front-side bus, up to 2GB of main memory, and up to 90GB of internal hard-disk space. Intel's Vpro cross-platform graphics family, delivering up to 64MB of DDR graphics memory, is making its first appearance in these machines. Prices for the new workstations start at $2,725 for the SGI 230.

SGI 230, 330, AND 550 VISUAL WORKSTATIONS | SGI | www.sgi.com

INDUSTRY WATCH

THE BUZZ ABOUT THE GAME BIZ | daniel huebner

Looking Glass Through. THIEF developer Looking Glass Studios has closed its doors. The decision was made in a company meeting, resulting in the cessation of operations and the cancellation of ongoing projects. Despite strong sales of THIEF and THIEF 2: THE METAL AGE, Looking Glass had been struggling financially. The company's hopes for an infusion of capital from publisher Eidos apparently fell victim of Eidos's own financial problems. Among the cancelled titles is THIEF 3. Though that game may hold interest for another publisher, designer Randy Smith expressed doubts in an open letter to fans that the game would ever see the light of day.

Nasdaq Cracks Down. The current malaise in the game industry is putting a number of publicly traded game companies in an uncomfortable situation. Nasdaq has threatened Interplay, Acclaim, and Infogrames with delisting. In all three cases, the companies have fallen below the requirements set by the exchange's Listing Qualifications Board, and appeals have been filed by all of the affected companies. Infogrames, which has enjoyed healthy returns despite recent unfavorable market conditions, contends that its failure to meet the $5 minimum bid requirement is simply the result of a stock split which is distorting its share price. Acclaim, meanwhile, is considering a stock consolidation to push its price back above the market's listing criteria.

Steed Ousted. Internal strife at id Software has led to the dismissal of modeler and animator Paul Steed. Commenting on the matter in his .plan file, id co-owner John Carmack suggested that his fellow co-owners Kevin Cloud and Adrian Carmack (no relation) fired Steed in direct retaliation over losing an internal dispute about the company's next title. Though a decision was made in favor of John Carmack's desire to create a third installment in the hit DOOM series, the fallout of the battle appears to be the loss of Steed. Cloud has disputed John Carmack's assertion, however, saying only that the impetus for Steed's dismissal went beyond any disputes over projects.

Sony Gears Up. Sony has clarified its online gaming goals with its acquisition of

The sequel to Looking Glass's dark-themed THIEF 2 may not be seeing the light of day.

EVERQUEST developer Verant Interactive. The privately held developer will become the new core of Sony Online Entertainment, with Sony phasing out its existing Los Angeles operations in order to consolidate its activities around Verant's San Diego headquarters. Sony has tapped Kelly Flock, who worked closely with Verant while heading up Sony's 989 Studios, as CEO of the new-look SOE. Sony is also preparing itself for the North American launch of Playstation 2 by investing more than a billion dollars to bolster Playstation 2 chip production. Some of the additional chips could find their way into machines other than the Playstation 2, as Sony has announced its intent to sell the new console's component chip to other manufactures in order to spread the platform's market share beyond home game machines.

Sega Keeps Up. Sega is keeping pace with Sony move for move as it prepares the Dreamcast to compete in the post–Playstation 2 console world. Sega has announced that it will beat Sony to the broadband punch by offering cable modem service to Dreamcast owners in Japan next month. The service will start in conjunction with 40 domestic cable television providers, and will eventually grow to include 200 providers. Sega will sell an adapter to connect the Dreamcast to cable line Internet systems. Sega also has plans in the works to sell game software directly to broadband users in the future.

In addition, he company is seeking ways to extend its platform beyond console gaming, and has teamed up with Motorola to create a Dreamcast-based cell phone. The two companies will jointly develop an API for the Internet-enabled phone using Dreamcast technology to allow fast downloads of games, images, and other data. Motorola hopes to use the technology to enhance its phones' data processing capacity as it looks to create next-generation applications including mobile video phones, while Sega is looking to diversify its product set by both producing games for the platform and collecting API license fees from Motorola. Sega's aggressive moves come in the wake of the company's third annual loss in a row, a loss that resulted in the resignation of Sega president Shoichiro Irimajiri. CSK chairman Isao Ohkawa has since taken on the job.

Nintendo Keeps Quiet. As Sony and Sega continue to fight for mindshare, Nintendo is still maintaining its low profile. The company has confirmed that it is delaying its next-generation Dolphin console until 2001, but Nintendo has still not been forthcoming about the console's specifics. Though Game Boy Advance will make its debut at Nintendo's Space World in Japan in August, Nintendo's E3 presence focused exclusively on existing products and platforms. It is hard to argue with the company's current success: though Nintendo's annual net profit fell 35 percent from last year, due largely to a overly strong yen, it still reached a healthy $521 million for the year.

UPCOMING EVENTS CALENDAR

ECTS

GRAND HALL
OLYMPIA CONVENTION CENTRE
London, England
September 3–5, 2000
Cost: variable
www.ects.com

FUN EXPO

SANDS EXPO AND CONVENTION CENTER
Las Vegas, Nev.
September 20–22, 2000
Cost: $10 and up
www.funexpo.com

Return to Cartoon Central

Adding Texture to a Nonphotorealistic Renderer

As I write this, I'm fresh back from E3, getting ready for Siggraph, and taking a week to go to the first international conference on nonphotorealistic rendering (NPR). I mean, how often do you get to take a trip through Burgundy to the French Alps and call it work?

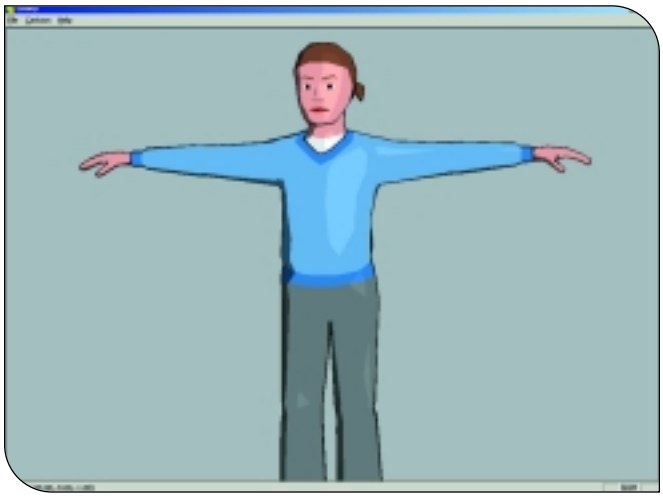
It all has me pretty energized for the advances I expect to happen in the game industry this year. While I didn't see many big surprises at E3, there was a lot to fuel my enthusiasm. Since my mind has been on NPR, I noticed several games that have embraced their limitations and gone for a more stylized look. There was certainly no shortage of racing games that are inspired by cartoons or license them directly. These titles traded realism for a 3D vision of the cartoon world. Games such as Sega's JET GRIND RADIO, Red Storm's ROSWELL CONSPIRACIES, and Kronos's FEAR EFFECT showed that you could use cartoon rendering techniques on detailed characters to achieve a very stylized look without sacrificing performance.

So this month I'm going to take another look at real-time 3D cartoon rendering and see how I can improve the look of my characters. So open up a nice bottle of Burgundy and let's journey...

Back to Toon Town

You may want to look back at my March 2000 column ("Shades of Disney: Opaquing in a 3D World") for a refresher. When we last left our flat-shaded friends, they were looking pretty toony with their 1D texture lookup table for shading and silhouette edge. You can see an example in Figure 1.

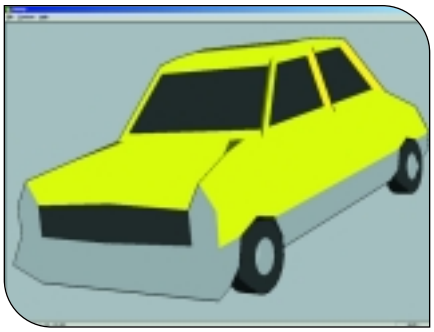
There are quite a few ways I can improve the quality of this character. The problem of cartoon rendering on a computer is very similar to that of traditional cartooning. There are two steps: generate the ink lines and paint the shade colors.



The ink lines come in several types. There is the silhouette edge, which defines the outline of the object. This is the most complicated to create as it is view-dependent and must be recalculated any time the camera or the object moves. The silhouette edge for this character was done easily using the hardware renderer, but it could still use some work. I will come back to that in a minute.

The second type of ink line commonly used in cartoon rendering is the material line. These lines separate materials in the object, making them stand out more. Material lines are easy to deal with. They are not view-dependent, which means I can calculate the material lines once and store them for rendering. The material lines can be detected using a preprocess pass through the mesh which finds adjacent triangles that use different materials. The edge that the triangles share is marked as a material edge to be rendered later.

The final type of ink line is known as the "hard" edge. This defines sharp creases in the model which give an object its distinctive look. The hard edge is also view-independent and, like the material line, can be calculated in a preprocess. In this case, the angle between two adjacent triangles is determined and if the angle is greater than



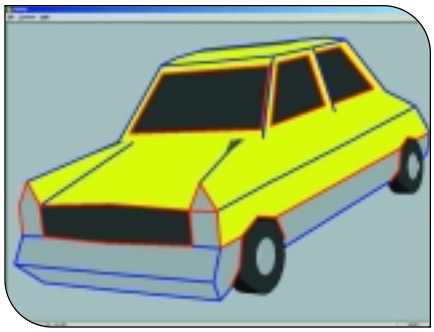


FIGURE 1 (top). A star from Toon Town, shaded with a 1D texture lookup table.
FIGURE 2A (bottom left). A car from Toon Town, sporting a view-dependent silhouette edge.
FIGURE 2B (bottom right). Here, the material edges are shown in red, the hard edges in blue. Both are view-independent and generally need to be calculated only once.

AUTHOR'S BIO | *Since Jeff has spent so much time thinking about stylized computer art we have started to wonder why he just doesn't pick up a pencil and start sketching instead. Let him know it's art not algorithms at jeffl@darwin3d.com.*

the crease threshold, the edge is marked as a hard edge. By default I'm going to define a hard edge as any crease greater than 30 degrees. That is, if the dot product of the two normals is greater than the cosine of 30 degrees, or 0.866, I will draw the hard edge. One important note is that if your object is deformable, the hard edges will change and therefore will need to be recalculated any time the character deforms.

You can see these two types of lines in Figures 2a and 2b. The red lines are material edges and the blue lines are hard edges.

The silhouette edge needs a little more care. If I want to improve the look of the silhouette much, I need to find the edges every frame (or at least anytime the model moves or the view changes). I can accomplish this by brute force, comparing each set of adjacent triangles to see if one faces toward the viewer while the other faces away. This is the requirement for a silhouette edge. Or, said mathematically:

$$\left(N_1 \bullet \left(V_1 - E\right)\right)\left(N_2 \bullet \left(V_2 - E\right)\right) \le 0$$

where N_1 and N_2 are the two face normals for the adjacent polygons, V_1 and V_2 are vertices on their respective edges, and E is the eye point. Whenever this statement is true, the edge is part of the silhouette.

I could just naively test every edge to see if it matches the dot product condition. Alternatively, I could test neighboring edges once I have found a silhouette edge and take advantage of any possible spatial coherence. A very interesting paper by Lee Markosian and his colleagues (see For More Information) describes possible optimizations of this silhouette edge detection.

Once all the edge types are detected, they can be rendered as lines or even textured polygons with some form of soft brush texture. An interesting approach for drawing the ink lines in a stylized manner has been proposed by J. D. Northrup and Lee Markosian (see For More Information). The technique stochastically varies the width of the ink lines and changes the color and opacity of the lines themselves. This creates much more natural-looking ink lines. I plan on investigating this technique further.

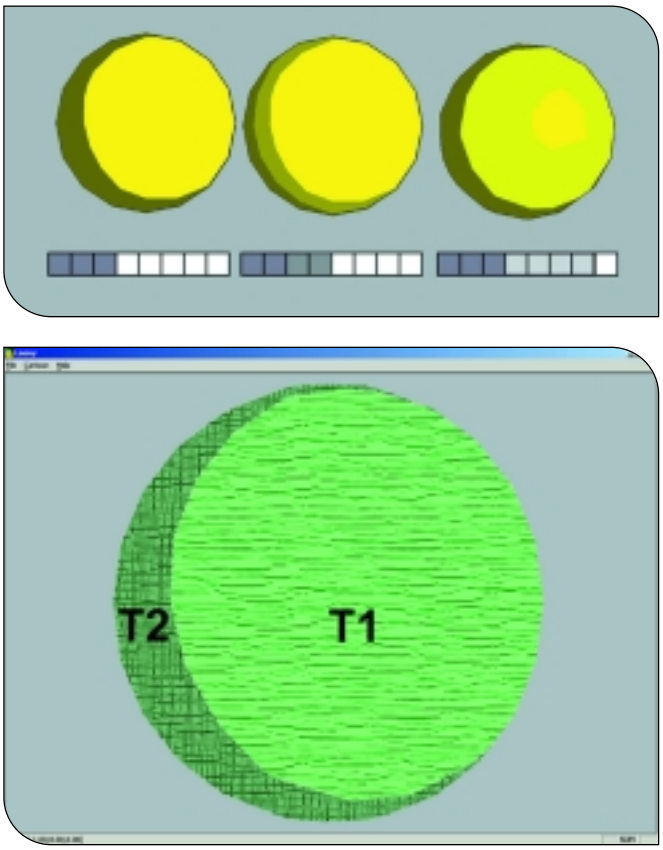


FIGURE 3 (top). The shade tables as 1D textures applied to a sphere.
FIGURE 4 (bottom). Two-texture shading.

Paint an Inch Thick

Now that the lines are all sorted out and looking nice, I want to do some work on the paint job. You may recall from my March column that in order to color my objects I used a 1D texture map as a nonlinear lookup table to determine when the shade should be applied. This shade is modulated with the object's material color to create the final image. To refresh your memory, Figure 3 shows various shade textures applied to a sphere.

That's pretty good for a basic cartoon look, but I want to make things a bit more stylized. Multi-pass graphics hardware is pretty common now among game players and it's a shame not to take advantage of it.

It may look interesting if I apply different stylized textures on the objects instead of using color shades. In fact, several people have done this already and it looks pretty good. Intel has developed a multi-pass method to apply textures to cartoon-shaded objects in their cartoon rendering system, which is available as part of Digimation's Real-Time 3D Libraries (see Product Update, April 2000). The only drawback to their method seems to be the large number of rendering passes needed to create the textured style.

In order to create a textured stylized rendering, I would like to use my shade tables to control texture rendering on an object. The goal would be to get something like you see in Figure 4, with one texture in the light area of the object and one in the dark area. It seems that would allow for a great deal of control over the placement of textures. From Figure 3, you can see that the shade table is implemented as grayscale colors that are modulated with the surface-material color in order to darken the shaded regions of the model.

However, there is no reason that this color needs to be grayscale. It could be any color that would then modulate with the surface color for different effects. Much more interesting is the fact that the shade table could contain alpha information. Imagine if the shade table under the first sphere were actually the alpha channel of the shade table color. All the dark areas in the shade table have alpha equal zero and all the light areas have alpha equal one.

Now here's where I get to use that cool multi-texture hardware you all have on your computers these days. Most game graphics cards since 3dfx's Voodoo 2 have been able to combine two textures at once in a technique called multi-texturing. Under OpenGL, I access this hardware feature by using the OpenGL extension `GL_ARB_multitexture` (or the Direct3D texture stage settings). I am going to use this

multi-texture along with two-pass rendering to get the image from Figure 4.

In pass one of the rendering, I will draw the entire object with the "dark" texture, T2. This will lead to some overdraw but we can optimize this later. Pass two is where the work gets done. This second pass uses the multi-texture mode of the hardware. There are now two textures that need to be specified. Texture one is the "light" texture, T1. Texture two will point to the 1D shade-table texture. Remember, my shade table contains the alpha color values. This pass is now rendered with alpha blending turned on. (Actually, alpha test is probably better since we are just using 1 and 0 for the alpha values as it may run faster than alpha blending on some hardware.)

What happens is that the shade-table texture coordinate is calculated just as I did for the basic cartoon rendering. The color result is multiplied with the texture color. Anywhere the shade table returns an alpha value of 0 will not render to the final display for that pass. Anywhere that the alpha value is 1 will get the new texture.

The final result is that I get the image in Figure 4. Of course it took two passes, one with multi-texture. But this would be a pretty tricky effect to achieve in other ways. Realize that the alpha values in the shade table do not need to be strictly 0 and 1. By using fractional values at any point in the shade table, I would get a proportional blend between the two textures.

By using this technique, I can create very stylized textures and selectively render them on an object to highlight its shape, much as a sketch artist does. You can see some objects with a pencil-sketch texture in Figure 5.

The main problem with this technique is that since it requires three textures, it requires the two passes. However, that's about to change. ATI recently announced its new 3D graphics chip, the Radeon 256, which will support blending of three simultaneous textures. That will allow me to draw a two-texture sketch-style object in a single rendering pass.

There are a lot of other interesting hardware developments that will really improve developers' abilities to create new stylized renderings. However, that will have to wait until next month. Till then, play with the new version of my cartoon renderer which now supports two stylized textures. Get the source and executable off the *Game Developer* web site at www.gdmag.com.

OpenGL Momentum Rant

Despite my best efforts most months, I'm generally not able to put together a long rant for this column. However, this month something has been troubling me enough to put the graphics aside and sound off. Professionally, I am reasonably API-agnostic. The final API target doesn't really matter that much as long as it does the job of exposing the hardware features I need. Making things work in an OpenGL or Direct3D final product is no big deal. For final delivery, it is speed and the hardware that matter, not the API. However, it must be clear to anyone who has read this column before that I have a fondness for OpenGL. This is for a variety of reasons.

I like the idea of a platform-neutral 3D rendering API. I would like to think that the technologies and ideas I create can work on a Macintosh, a Linux box, SGI, or any other system that supports accelerated 3D rendering. The fact that many of my demos have been converted to a variety of platforms proves that to some extent. There would probably be more of this if I were less lazy about UI creation and relied less on Windows controls. However, trade-offs need to be made between ease of use and platform-independence.

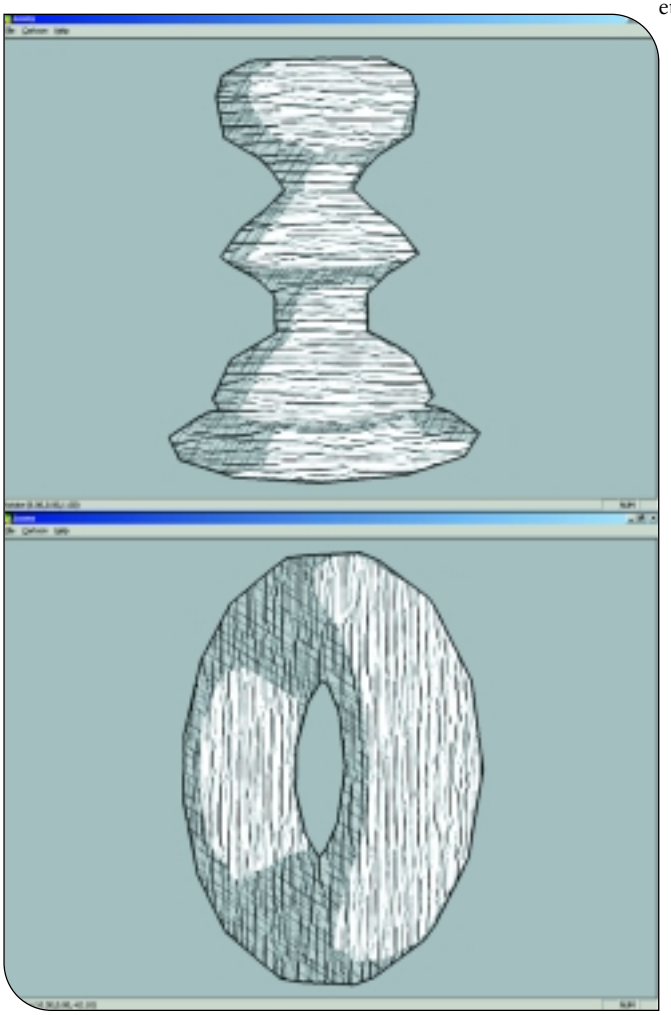


FIGURE 5. Dual-texture sketch-style rendering.

I also find OpenGL to be an incredibly creative API. I can get something running and experiment with it very easily. This is similar to an artist who finds a particular animation package conducive to his or her creativity. Most high-end animation programs can yield nice images, however, all artists have their favorite. I happen to find OpenGL a very creative and intuitive graphics programming API for both research and education.

Having explained all that, I find myself troubled by the state of OpenGL. This has been simmering for some time now and doesn't show any signs of getting better. For a long time, OpenGL took a leadership role in real-time graphics rendering on hardware platforms. Consumer-level 3D hardware was in its infancy and looked to the CAD and simulation markets for inspiration. They had flashy features such as geometry acceleration, filtered

texturing, and stencil buffers. We had hardware that could hardly render a textured triangle.

Well, times have changed dramatically. The consumer 3D graphics market has advanced at a feverish pace. Competition between hardware vendors led to feature after feature and the list of features we lacked quickly grew shorter. At the same time, game developers began to recognize that there were things important to us that were not important to OpenGL's traditional users up until that point. Texturing with multiple rendering passes became a must-have feature for graphics hardware. Clever ways to do real-time environment and bump maps were needed. Developers began looking to the OpenGL programming manuals for inspiration for new effects and found none.

So, developers started to read Siggraph proceedings and books like *The RenderMan Companion* looking for inspiration. Hardware vendors started asking what kinds of new features we desired. We passed the last road sign directing us down the real-time rendering highway.

The problem is that software developers and hardware manufacturers are fundamentally at conflict. Hardware makers want to sell new products every year. They want to develop features that differentiate themselves from their competition and give the buyer a reason to select their product. Software makers, on the other hand, want the best collection of features that work on the largest set of systems. Game developers would gladly sacrifice vendor-specific features for cross-system stability. These two philosophies are in direct opposition.

There needs to be an arbiter of these conflicting motivations. This arbiter must combine the information and ideas from all sides involved and develop a plan for the future of real-time 3D graphics. Microsoft is in a perfect position for this role, and as you know, has stepped in to fill it. They discuss with developers their needs and wants (really, they do) and talk to the hardware manufacturers to find out what is possible. They then plot the road map for where 3D hardware should go and design an API to give developers access to these features. This is where the arguments start.

Microsoft is not really interested in developing OpenGL. I understand that. They are interested in designing an API for gaming on Windows. Hopefully, they will make it the easiest API possible for developers to use. Hopefully, they listen to what developers say they want to see in an API and implement it. Whether you use it in your game or not is your decision, just like you decide which animation package to use.

But in its role as real-time rendering visionary, Microsoft actually does a lot more than define an API that developers can use. They actually guide future hardware development to some extent. They set goals and targets for hardware makers and in doing so define many of the features exposed to developers. This affects all APIs and all operating systems that may use this hardware.

Unfortunately for OpenGL, there seems to be no one carrying a vision for its future.

Unfortunately for OpenGL, there seems to be no one carrying this vision for its future. There is no one really working between developers and vendors to define the path. There is the OpenGL Architecture Review Board (ARB), which governs the official OpenGL specification. However, the ARB encompasses a wide variety of interests including the manufacturing, simulation, and defense industries. It is a very conservative body that implements well-thought-out features, but this takes time. Without this vision, OpenGL is left to implement extensions that give access to the features created by the D3D vision. OpenGL support is left as an orphaned afterthought.

So who could carry this vision and how would it fit in with OpenGL? SGI once carried the vision for the future of rendering. People like Mark Peercy and others continue to come up with very interesting ideas at the company. Many also point to Nvidia as a potential standard-bearer of OpenGL innovation. This makes some sense as many of the best minds that worked on OpenGL now work for that company. However, hardware vendors do not tend to look to each other for areas in which to innovate. As I said before, they are looking to differentiate themselves to consumers, not come up with a common direction for development.

This leaves the software developers themselves. We need to spend some time focusing on the features future graphics hardware should have. We need to consider how we want to access these features and we need to work with the hardware vendors to make sure it can happen. Developers need to become more active in API design instead of just grumbling about how difficult a given one is to use or how extensions are not widely supported. This currently happens to some extent through private conversations, .plan files, and newsgroup postings. However, we really need to organize these discussions in order to present a coherent plan to hardware vendors. Getting the features done right in hardware is the key. That means we need to try ideas out in software instead of on the next graphics card release. The OpenGL extension system is an ideal way to experiment with new, forward-thinking ideas before things get locked into silicon. Forums like the Game Developers Conference, the GDC Hardcore Technical Seminars, and public newsgroups and mailing lists are perfect vehicles for developers to get involved. We need to discuss what we want to see and how we want to use it. The future of your favorite API depends on it.

FOR MORE INFORMATION

Markosian, Lee, and others. "Real-Time Nonphotorealistic Rendering." *Proceedings of Siggraph 97*. pp. 415–420.

Northrup, J. D., and Lee Markosian. "Artistic Silhouettes: A Hybrid Approach." *Proceedings of NPAR 2000* (forthcoming).

Digimation Real-Time 3D Libraries
www.digimation.com

Bump and Shine

Getting Down with New 3D Hardware Effects

There have been some major changes in my little real-time 3D world lately, and I have a feeling that there are many more to come. I used to pride myself on pinching every last vertex and polygon until my model had just enough detail to tell you its form. Now I'm using vast arrays of polygons just to get the shading effect that I want. I used to spend hours painting specular highlights and shadows into tiny textures, grooming each pixel to add to the illusion. Now I spend that time creating detail and subtlety on high-resolution textures that don't need much lighting information at all. I'm using new modeling techniques and creating special textures for effects that I never thought would be possible in real time. What have I been working on, you ask? Demos, I'll tell you. Demos to illustrate some of the things that can be done with the new features of the latest 3D graphics chips.

There are a lot of new possibilities for what can be done as hardware technology marches along, and as artists and programmers work with the new technologies a lot of new effects will be discovered. I can't tell you all there is to know about what effects are now possible, but I can tell you about three very interesting ones: real-time reflection, masked real-time reflection and real-time bump mapping. These are all effects that have only recently become possible in real time thanks to advances in graphics hardware, and have already cropped up in games such as EVOLVA. The demo that I was involved in to show off these nifty new effects consisted of a 22,000-polygon 3D model of the graphics chipmaker's company logo and several different sets of textures to showcase each effect. The simplicity of the art production and the conciseness of what the demo was designed to illustrate makes it a good jumping-off point for explaining what some of this new technology means for artists.

Real-Time Reflection

Real-time reflection can be amazing to behold. As you might guess, as you rotate an object that has a reflective material on it, say a polished metal ball, the reflection changes to give the illusion that the object is reflecting back the correct part of the environment (Figure 1). Making objects reflective in real time is a key element in making computer art look lifelike, as almost everything in the real world has some reflective properties. Even specular highlight is a form of reflection, you can imagine the added realism of having it move as you rotate an object. So, what is needed from the artist to create real-time reflection? There are three basic elements: a model that is built with certain issues in mind (more on that later), a base texture, and a reflection map. One other element that's worth mentioning is a good working relationship with the programmer who will be implementing the new features. As with the implementation of any new feature, and particularly because these new effects happen in the engine (as opposed to being painted into a texture), setting up a clear procedure with the programmers on your project will get you closer to the results that you want in the end.

The reflective model. As I mentioned, there are certain issues that you need to keep in mind when you are building a model that's going to use real-time reflection, especially objects with a highly reflective surface such as polished metal. The main issue is that the model needs to have an evenly spaced and hopefully plentiful array of vertices across the reflective surface. These vertices are necessary in order to get the most realistic and accurate reflection. The reflection is calculated based on the normals of the vertices and the angle of

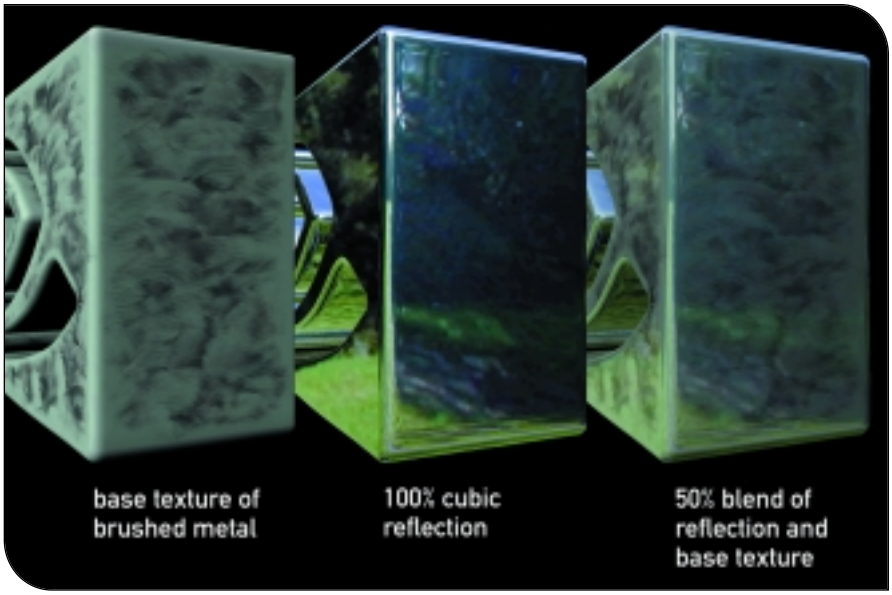


FIGURE 1. Real-time reflection on an object, utilizing both a base texture and a reflection map.

AUTHOR'S BIO | *Lisa Washburn's bumpy and shiny new world reflects Vector Graphics (www.vectorg.com), her real-time 3D art production company. Send comments and questions to article@vectorg.com.*

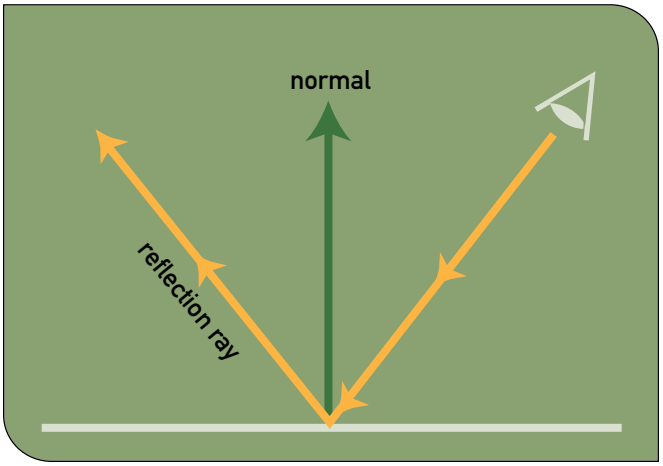


FIGURE 2 (above). Calculating the reflection. More vertices produce better results and the technique lends itself well to patch-based objects.
FIGURE 3 (below). Using spherical reflection maps can produce appealing results, but distort when the camera or viewpoint changes.

the camera (or viewer), as you can see in Figure 2. What gets reflected between the vertices is interpolated from this information, so the more vertices you have, the less guesswork the computer has to do, and the more accurate your reflections will be. Luckily, advances in graphics cards combined with faster CPU processing speeds give you a lot more vertices to play with. I found that building objects with patches can give you a nice even grid of vertices when you change it into a polygonal model (see "Skin Deep 2: Implementing Patch Surfaces," Artist's View, April 2000, for more information on working with patches).

The base texture. For the demo mentioned earlier, which I will refer to as the logo demo, we decided to go for a brushed-metal effect. The most interesting thing about texturing this model, and one of the biggest changes from nonaccelerated real-time 3D, is that there was no need to paint lighting into it. The whole point of real-time reflection is that the texture changes as you move the object around, so painted-on specular highlights would look obviously static compared with the dynamic reflection. New features such as stencil buffering give you several options for dynamic shadows. This particular demo didn't feature dynamic shadows, so that will have to wait for another article. The other interesting thing about the textures was their enormous size. We used nine textures, two at 1024×1024 and seven at 1024×512, on this part of the demo alone. It probably would have been possible to use more, but that's all that was really necessary for the effect that we wanted. Granted we only had one object to texture, so depending on your scene you will use different configurations of texture sizes, but that's a lot of texture given all the other effects that were being processed.

Reflection map. Calculating reflections is a bit like ray-tracing. You bounce a ray off of the object and, based on a piece of math that uses the angle at which you are looking at the object and the normal of the nearest vertex, you project the ray out into the scene. Where the ray hits is what is reflected on your object. Imagine the math that would have to be processed if the computer had to bounce a ray off the object for every pixel that it drew and also had to detect where in the scene the ray went and what it was hitting. After detecting what the ray hit, it would have to figure out what that should look like and then render it in the reflection. That's a lot of calculations.

Enter the third piece of art that is needed to create real-time reflections: the reflection map. A reflection map is a 2D image of the scene surrounding the shiny object. This drastically optimizes the rendering time by simplifying the calculations done by the computer to figure out what to reflect. Using a reflection map, the computer calculates a second set of UV coordinates that tells it where on the reflection map to render, instead of the computer having to detect where in the scene the ray has bounced to. This calculation is done for each vertex that is rendered. As I mentioned when I discussed building the model, the area between vertices is interpolated from the information at the vertices. Reflection maps are a bit more complicated than just a 2D picture, however. There are two different types of reflection maps that are used most commonly in real-time reflection, namely spherical and cubic. There is a third option called planar reflection, but this only works for flat surfaces and is generally calculated on the fly. A fourth option is dual-paraboloid environment mapping, but it is not commonly used and therefore will not be covered here.

A spherical reflection map is a single 2D image that looks like you took the reflective object, say a drop of water, surrounded it with a shiny chrome ball and then took a picture of the chrome ball. Basically it's a picture of the environment surrounding the droplet reflected in a chrome ball, an example of which you can see in Figure 3 where a bee is standing next to a drop of water on a leaf. There are two major drawbacks to using spherical maps.

The first is that it is only truly accurate from the viewpoint at which the reflection image was generated. As you move farther away from that viewpoint the distortion gets worse, especially along the edges and in particular around the back. This means that if you are using a spherical map you can rotate the shiny object, but as soon as you move the camera the reflection will be distorted, and possibly completely inaccurate. For example, imagine that you are holding a chrome ball in front of your face. If you turn the ball, without moving your head, your face with the same background will be reflected in the side of the ball that is facing you. The viewpoint hasn't changed. If that same chrome ball is spherically mapped in a real-time 3D environment, and you move the camera around to look at the back, you will see distortion that looks like the reflection has been bunched together to cover a hole. It's similar to the mapping distortion that you get at the

poles of a ball when you use the spherical mapping option in your 3D modeling software. Objects that are only somewhat reflective can probably get away with this type of distortion. The shinier the reflection, however, the more obvious the inaccuracies will be.

The second problem is that because of the way in which the map needs to be distorted, spherical maps are difficult to generate and aren't usually generated on the fly in the game engine. You take a rendering of the environment, use the Spherize plug-in in Photoshop, and hand-paint the specular highlights to create something like what you see in Figure 3. It's not strictly accurate from a mathematical standpoint, but it still gives you a reasonable spherical reflection effect.

The second kind of environment map, which we used for the logo demo, is cubic mapping. Cubic mapping uses six different images that are generated by taking your shiny object and surrounding it with a cube. Now put a camera in place of the shiny object and render six images of the surrounding environment using each side of the cube as its frame. The camera rotates in 90-degree increments, but doesn't move. You will get an image for front, back, right, left, top, and bottom, as you can see in Figure 4. The biggest advantage to cubic mapping is that it is viewpoint-independent. Unlike the spherical map that was only accurate from the viewpoint from which it was rendered, cubic mapping references six different viewpoints. This means that you can rotate the camera in the scene, or the object, and you will still get an accurate reflection. Cubic reflection maps are also easy to generate, as they are straightforward renderings of the environment, as opposed to the chrome ball effect of the spherical map. The only stipulations for cubic maps is that they need to have a 1:1 aspect ratio, have a 90-degree field of view, and be pixel-tight with no overlapping at the edges. If you have a digital environment already built you can use your 3D modeling package to generate them. 3D Studio Max has a Reflect/Refract Map option in the material editor that will generate a set of cubic maps for you. Other software packages have similar options.

If you want to reflect an actual physical environment there are even digital cameras that can be set up to generate cubic maps. These are used for "virtual tour"–type options on the Internet. Figure 4, which was used in the logo demo, was generated using a digital camera. If you have the bandwidth, cubic reflection maps can also be generated on the fly by setting up the six extra cameras in your scene and having them render to the reflection. This gives you the option of capturing moving objects in your reflection maps. The biggest drawbacks to cubic reflection maps are the logistics of handling six textures instead of the single one that spherical mapping uses, and that not all real-time 3D platforms support them.

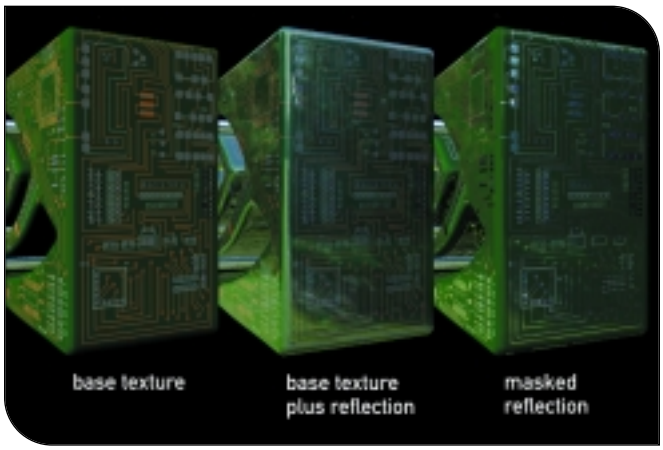


FIGURE 4 (top). A cubic environment map utilizes six different viewpoints, enabling the object or viewpoint to rotate without distortion.
FIGURE 5 (bottom). Masked real-time reflection uses the base texture's alpha channel to mask certain areas of an object's reflective surface.

Masked Real-Time Reflection

Masked real-time reflection uses the alpha channel of the color texture map to control the level of reflection for the shiny object. Using the alpha channel to control reflection is very similar to using it to control transparency. By using a gradient from black to white, you can specify how shiny an object will be when it is rendered. For example, if you specify that black is mirror-shiny and white is not reflective at all, then a light gray would be a dull shine similar to lightly polished marble. Likewise, you can also mask out areas so that there are shiny details inside of nonreflective textures.

This is what we did for the logo demo. I created a circuit-board design that was green with metal traces and soldering joints running across it. I masked out the green areas, which created the effect of shiny metal details on a matte plastic background (see Figure 5). This technique opens up all kinds of opportunities. Shiny objects can have dull, scuffed areas, such as rub marks or fingerprints on a doorknob, or scratches on the fender of a car. Semi-reflective objects can have highly reflective details such as a gold inlay on the polished black handle of a sword or any other number of possibilities.

Real-Time Bump Mapping

Bump mapping is used to create the illusion of 3D detail on a surface. It creates the illusion through a series of 2D textures and doesn't require the detail to

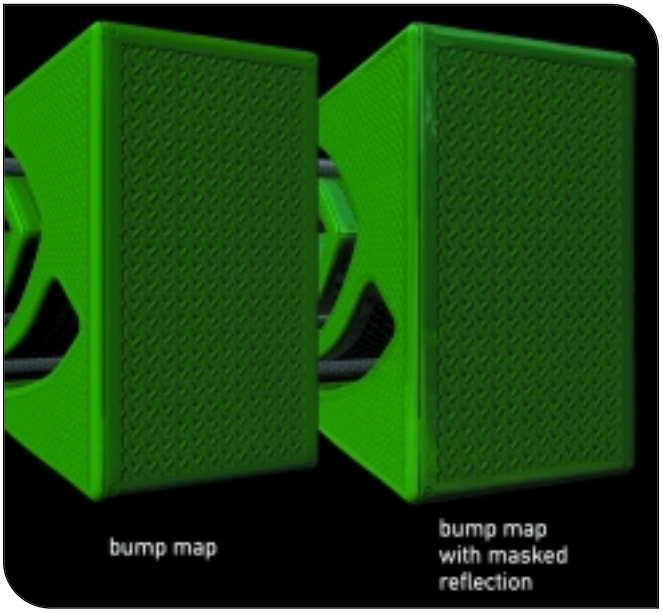


FIGURE 6 (top). Real-time bump mapping.
FIGURE 7 (bottom). The grayscale map used.

be present in the geometry. It adds realism to objects that have a bumpy, wrinkled, or cracked surface by creating the look of subtle shadow changes as a light source is moved around it. It's a recent addition to real-time 3D, although it's been used in prerendered 3D for years.

There are several different ways to do real-time bump mapping, each with varying degrees of realism and performance hit. The procedure that we used for the logo demo is generally called "dot product 3," which refers to the type of calculation used to figure out the lighting. On the art production side of bump mapping there are two things that are needed: the color texture and a grayscale image (essentially a height map) that determines what the bump looks like. This grayscale image can be in the alpha of the color texture, or it can be a separate image. For the logo demo, we didn't use a color texture, just the bump map and reflection, so the bump map was not in the alpha channel. You can see the results in Figure 6. The grayscale map was of textured steel plating with a metal border around the edge and screws in the corners (Figure 7).

To create the effect of real-time bump mapping, the grayscale height map is turned into a normal map for each pixel of the object. This means that each pixel of the object now has a directional normal based on its shade of gray. Next, a vector is generated that points to the light source in the scene. This pixel normal and the light vector are then used to calculate where the highlight and shadow should be. As the light is moved around the object, the shadow and highlight are recalculated for each pixel. This gives you the effect of light playing over the surface of a bumpy object, even though there is no geometry there to interact with the light.

A tip for creating color maps that have companion bump maps: don't put any lighting in the color map. For example, say that your object is a rock with a crack in it. It's O.K. to add details to the crack, such as accumulated dirt, but you will get better results if you don't put in a dark shadow. If the crack is specified in the bump map, then the shadow will be generated dynamically as a light is passed over it. If the shadow is already statically painted into the color map, then the bump map will not look right.

Real-time 3D is growing closer to prerendered 3D with every turn of the graphic chip development cycle. What does this mean for artists? It means that we have to break out those tutorials and explore how to use some of the more advanced features of our 3D software that we thought we wouldn't ever get to use. We artists will soon spend our time actually building the detail into our models instead of trying to figure out ways to fake them, be it through texture or modeling tricks. We will actually light scenes instead of figuring out ways to paint lighting effects into the texture or light map. We will be freed up to use more of the dynamic material options that ship with our 3D package of choice, like bump mapping and cubic reflection. And finally, it means that with the help of a talented programmer, RT3D artists are poised on the brink of creating some of the most realistic, mind-blowing 3D artwork in the history of computer gaming graphics. Well, at the very least some of the most bumpy and shiny.

FOR MORE INFORMATION

There's a lot of information on the developer pages of all the chip manufacturers.

NVIDIA
www.nvidia.com/Developer.nsf

ATI
www.ati.com/na/pages/resource_centre/dev_rel/devrel.html

3DFX
www.3dfx.com or e-mail devprogram@3dfx.com

ACKNOWLEDGEMENTS

Special thanks to Curtis Beeson and Joe Demers, Jeff Lander of Darwin 3D (www.darwin3d.com), and Chris Hecker of definition six (www.d6.com) for their technical insights and huge amounts of patience.

Return

GAME AI

The State of the Industry

by steven woodcock

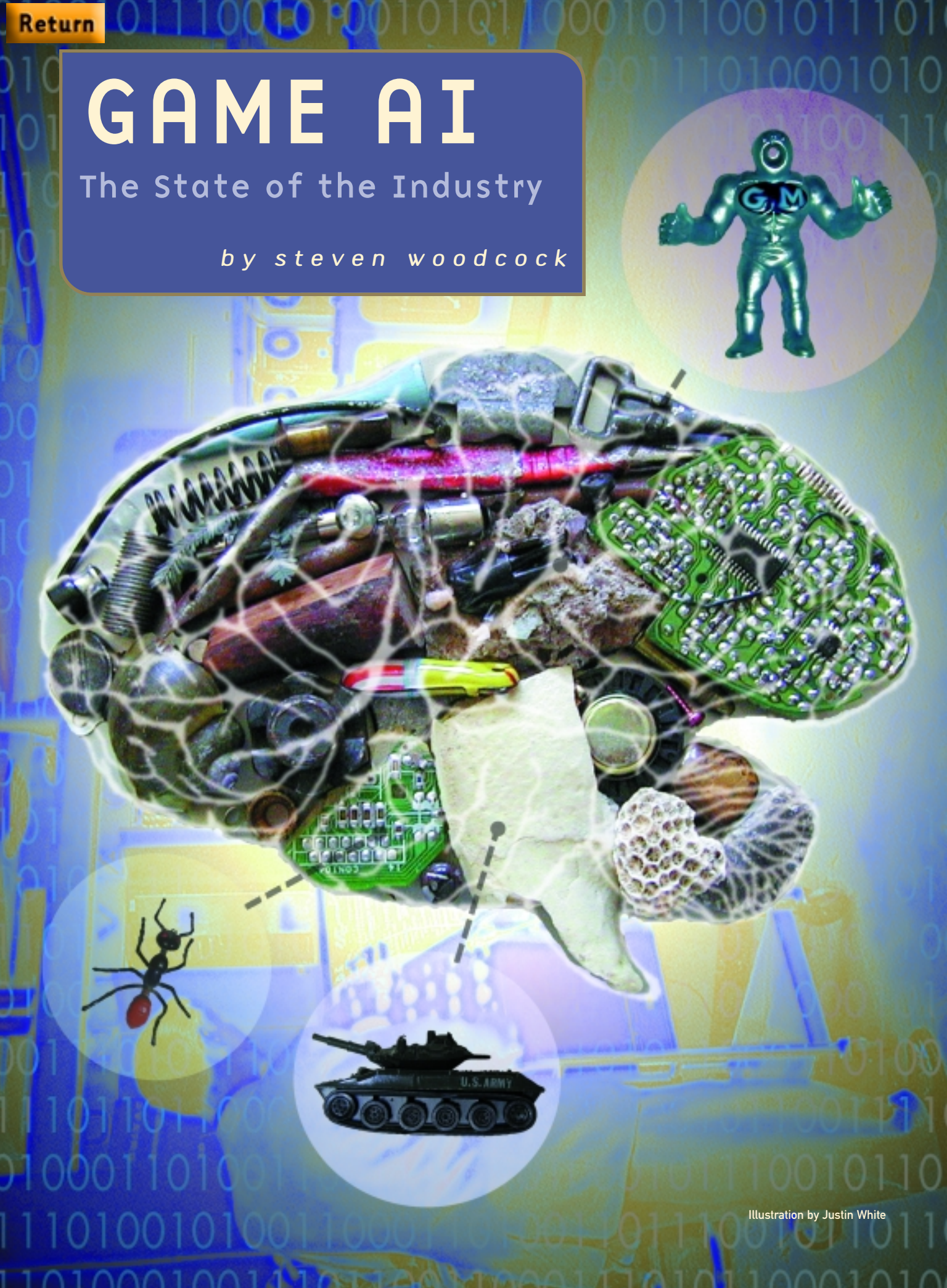


Illustration by Justin White

One thing was made clear in the aftermath of this year's Game Developers Conference: game AI has finally "made it" in the minds of developers, producers, and management. It is recognized as an important part of the game design process. No longer is it relegated to the backwater of the schedule, something to be done by a part-time intern over the summer. For many people, crafting a game's AI has become every bit as important as the features the game's graphics engine will sport. In other words, game AI is now a "checklist" item, and the response to both our AI roundtables at this year's GDC and various polls on my game AI web site (www.gameai.com) bear witness to the fact that developers are aggressively seeking new and better ways to make their AI stand out from that of other games.

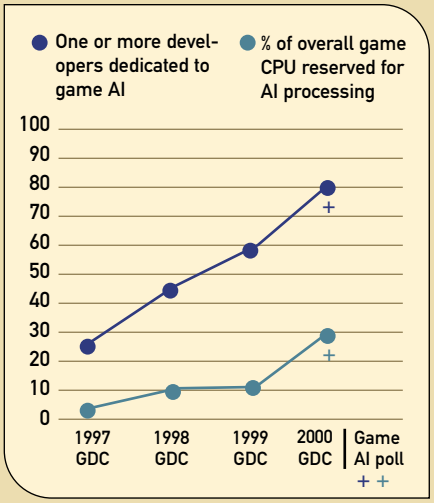


FIGURE 1. **AI poll results from the GDC 2000 roundtables.**

The technical level and quality of the GDC AI roundtable discussions continues to increase. More important, however, was that our "AI for Beginners" session was packed. There seem to be a lot of developers, producers, and artists that want to understand the basics of AI, whether it's so they can go forth and write the next great game AI or just so they can understand what their programmers are telling them.

As I've done in years past, I'll use this article to touch on some of the insights I gleaned from the roundtable discussions that Neil Kirby, Eric Dybsand, and I conducted. These forums are invaluable for discovering the problems developers face, what techniques they're using, and where they think the industry is going. I'll also discuss some of the poll results taken over the past year on my web site, some of which also provided interesting grist for the roundtable discussions.

Resources: The Big Non-issue

Last year's article ("Game AI: The State of the Industry," August 1999) mentioned that AI developers were (finally) becoming more involved in the game design process and using their involvement to help craft better AI opponents. I also noted that more projects were devoting more programmers to game AI, and AI programmers were getting a bigger chunk of the overall CPU resources as well.

This year's roundtables revealed that, for the most part, the resource battle is over (Figure 1). Nearly 80 percent of the developers attending the roundtables reported at least one person working full-time on AI on either a current or previous project; roughly one-third of those reported that two or more developers were working full-time on AI. This rapid increase in programming resources has been evident over the last few years in the overall increase in AI quality throughout the industry, and is probably close to the maximum one could reasonably expect a team to devote to AI given the realities of the industry and the marketplace.

Even more interesting was the amount of CPU resources that developers say they're getting. On average, developers say they now get a whopping 25 percent of the CPU's cycles, which is a 250 percent increase over the average amount of CPU resources developers said they were getting at the 1999 roundtables. When you factor in the increase in CPU power year after year, this trend becomes even more remarkable.

Many developers also reported that general attitudes toward game AI have shifted. In prior years the mantra was "as long as it doesn't affect the frame rate," but this year people reported that there is a growing recognition by entire development teams that AI is as important as other aspects of the game. Believe it or not, a few programmers actually reported the incredible luxury of being able to say to their team, "New graphics features are fine, so long as they don't slow down the AI." If that isn't a sign of how seriously game AI is now being taken, I don't know what is.

Developers didn't feel pressured by resources, either. Some developers (mostly those working on turn-based games) continued to gleefully remind everyone that they devoted practically 100 percent of the computer's resources for computer-opponent AI, but they also admitted that this generally allowed deeper play, but not always better play. (It's interesting to note that all of the turn-based developers at the roundtables were doing strategy games of some kind — more than other genres, that market has remained the most resistant to the lure of real-time play.) Nearly every developer was making heavy use of threads for their AIs in one fashion or another, in part to better utilize the CPU but also often just to help isolate AI processes from the rest of the game engine.

AI developers continued to credit 3D graphics chips for their increased use of CPU resources. Graphics programmers simply don't need as much of the CPU as they once did.

AUTHOR'S BIO | *Steven Woodcock's background in game AI comes from 16 years of ballistic missile defense work building massive real-time war-games and simulators. He did a stint in the consumer arena, then returned to the defense world to help develop the AI for the national missile defense system. He maintains a web site dedicated to game AI at www.gameai.com, and is the author of a number of papers and publications on the subject. Most recently, he contributed to a chapter to* Game Programming Gems *(Charles River Media, 2000). Steve lives in gorgeous Colorado Springs, Colo., at the foot of Pikes Peak with his lovely wife Colleen and an indeterminate number of pet ferrets. His hobbies include hiking, shooting, writing, and anything* Battlestar: Galactica *(go figure). Contact him at ferretman@gameai.com.*

Trends Since Last Year

A number of AI technologies noted at the 1998 and 1999 GDCs has continued to grow and accelerate over the last year. The number of games released in recent months that emphasize interesting AI — and which actually deliver on their promise — is a testament to the rising level of expertise in the industry. Here's a look at some trends.

Artificial life. Perhaps the most obvious trend since the 1999 GDC was the wave of games using artificial life (A-Life) techniques of one kind or another. From Maxis's THE SIMS to CogniToy's MIND ROVER, developers are finding that A-Life techniques provide them with flexible ways to create realistic, lifelike behavior in their game characters.

The power of A-Life techniques stems from its roots in the study of real-world living organisms. A-Life seeks to emulate that behavior through a variety of methods that can use hard-coded rules, genetic algorithms, flocking algorithms, and so on. Rather than try to code up a huge variety of extremely complex behaviors (similar to cooking a big meal), developers can break down the problem into smaller pieces (for example, open refrigerator, grab a dinner, put it in the microwave). These behaviors are then linked in some kind of decision-making hierarchy that the game characters use (in conjunction with motivating emotions, if any) to determine what actions they need to take to satisfy their needs. The interactions that occur between the low-level, explicitly coded behaviors and the motivations/needs of the characters causes higher-level, more "intelligent" behaviors to emerge without any explicit, complex programming.

The simplicity of this approach combined with the amazing resultant behaviors has proved irresistible to a number of developers over the last year, and a number of games have made use of the technique. THE SIMS is probably the best known of these. That game makes use of a technique that Maxis co-founder and SIMS designer Will Wright has dubbed "smart terrain." In the game, all characters have various motivations and needs, and the terrain offers various ways to satisfy those needs. Each piece of terrain broadcasts to nearby characters what it has to offer. For example, when a hungry character walks near a refrigerator, the refrigerator's "I have food" broadcast allows the character to decide to get some food from it. The food itself broadcasts that it needs cooking, and the microwave broadcasts that it can cook food. Thus the character is guided from action to action realistically, driven only by simple, object-level programming.

Developers were definitely taken with the possibilities of this approach, and there was much discussion about it at the roundtables. The idea has obvious possibilities for other game genres as well. Imagine a first-person shooter, for example, in which a given room that has seen lots of frags "broadcasts" this fact to the NPCs assisting your player's character. The NPC could then get nervous and anxious, and have a "bad feeling" about the room — all of which would serve to heighten the playing experience and make it more realistic and entertaining. Several developers took copious notes on this technique, so we'll probably be seeing even more A-Life in games in the future.

TOP. A smart rover navigates a maze in CogniToy's MIND ROVER.
BOTTOM. THE SIMS made ample use of A-Life technology.

Pathfinding. In a remarkable departure from the roundtables of previous years, developers really didn't have much to ask or say about pathfinding at this year's GDC roundtables. The A* algorithm (for more details, see Bryan Stout's excellent article "Smart Moves: Intelligent Path-Finding," October/November 1996) continues to reign as the preferred pathfinding algorithm, although everybody has their own variations and adaptations for their particular project. Every developer present who had needed pathfinding in their game had used some form of the A* algorithm. Most had also used influence maps, attractor-repulsor systems, and flocking to one degree or another. Generally speaking, the game community has this problem well in hand and is now focusing on particular implementations for specific games (such as pathfinding in 3D space, doing real-time path-granularity adjustments, efficiently recognizing when paths were blocked, and so on).

As developers become more comfortable with their pathfinding tools, we are beginning to see complex pathfinding coupled with terrain analysis. Terrain analysis is a much tougher problem than simple pathfinding in that the AI must study the terrain and look for various natural features — choke-points, ambush locations, and the like. Good terrain analysis can provide a game's AI with multiple "resolutions" of information about the game map that are well tuned for solving complex pathfinding problems. Terrain analysis also helps make the AI's knowledge of the map more location-based, which (as we've seen in the example of THE SIMS) can simplify many of the AI's tasks. Unfortunately, terrain analysis is made somewhat harder when randomly generated maps are used, a feature which is popular in today's

games. Randomly generating terrain precludes developers from "pre-analyzing" maps by hand and loading the results directly into the game's AI.

Several games released in the past year have made attempts at terrain analysis. For example, Ensemble Studios completely revamped the pathfinding approach used in AGE OF EMPIRES for its successor, AGE OF KINGS, which uses some fairly sophisticated terrain-analysis capabilities. Influence maps were used to identify important locations such as gold mines and ideal locations for building placement relative to them. They're also used to identify staging areas and routes for attacks: the AI plots out all the influences of known enemy buildings so that it can find a route into an enemy's domain that avoids any possible early warning.

Another game that makes interesting use of terrain analysis is Red Storm's FORCE 21. The developers used a visibility graph (see "Visibility Graphs," p. 30) to break down the game's terrain into distinct but interconnected areas; the AI can then use these larger areas for higher-level pathfinding and vehicle direction. By cleanly dividing maps into "areas I can go" and "areas I can't get to," the AI is able to issue higher-level movement orders to its units and leave the implementation issues (such as not running into things, deciding whether to go over the bridge or through the stream, and so on) to the units themselves. This in turn has an additional benefit: the units can make use of the A* algorithm to solve smaller, local problems, thus leaving more of the CPU for other AI activity.

Formations. Closely related to the subject of pathfinding in general is that of unit formations — techniques used by developers to make groups of military units behave realistically. While only a few developers present at this year's roundtables had actually needed to use formations in their games, the topic sparked quite a bit of interest (probably due to the recent spate of games with this feature). Most of those who had implemented formations had used some form of flocking with a strict overlying rules-based system to ensure that units stayed where they were supposed to. One developer, who was working on a sports game, said he was investigating using a "playbook" approach (similar to that used by a football coach) to tell his units where to go.

TOP. Ensemble Studios revamped the pathfinding in AGE OF EMPIRES II: THE AGE OF KINGS by including terrain analysis.
BOTTOM. Red Orb's PRINCE OF PERSIA 3D used The Motion Factory's Motivate SDK.

State machines and hierarchical AIs. The simple rules-based finite- and fuzzy-state machines (FSMs and FuSMs) continue to be the tools of choice for developers, overshadowing more "academic" technologies such as neural networks and genetic algorithms. Developers find that their simplicity makes these approaches far easier to understand and debug, and they work well in combination with the types of encapsulation seen in games using A-Life techniques.

Developers are looking for new ways to use these tools. For many of the same reasons A-Life techniques are being used to break down and simplify complex AI decisions into a series of small, easily defined steps, developers are taking more of a layered, hierarchical approach to AI design. Interplay's STARFLEET COMMAND and Red Storm's FORCE 21 take such an approach, using higher-level strategic "admirals" or "generals" to issue general movement and attack orders to tactical groups of units under their command. In FORCE 21 these units are organized at a tactical level into platoons; each platoon has a "tactician" who interprets the orders the platoon has received and turns them into specific movement and attack orders for individual vehicles.

Most developers at the roundtables who were working on strategy games reported that they were either planning to implement or already had used this type of layered approach to their AI engines. Not only was it a more realistic representation, but it made debugging simpler. Most of those who used this design also liked the way it allowed them to add hooks at the strategic level to allow for user customization of AIs, building strategies, and so on, while isolating the lower-level "get the job done" AI from anything untoward that the user might accidentally do to it. This is another trend we're seeing in strategy games that players find quite enjoyable — witness the various "empire mods" for games such as STARS, EMPIRE OF THE FADING SUNS and ALPHA CENTAURI.

Can AI SDKs Help?

The single biggest topic of discussion at the GDC 2000 roundtables was the feasibility of AI SDKs. There are at least three software development kits currently available to AI developers:

- Mathématiques Appliquées' DirectIA, an agent-based toolkit that uses state machines to build up emergent behaviors.
- Louder Than A Bomb's Spark!, a fuzzy-logic editor intended for AI engine developers.
- The Motion Factory's Motivate, which can provide some fairly sophisticated action/reaction state machine capabilities for animating characters. It was used in Red Orb's PRINCE OF PERSIA 3D, among others.

Many developers (especially those at the "AI for Beginners" session) were relatively unaware of these toolkits and hence were very interested in their capabilities. It didn't seem, however, that many of the more experienced developers thought these toolkits would be all that useful, though a quick poll did reveal that one or two developers were in the process of evaluating the DirectIA toolkit. Most expressed the opinion that one or more SDKs would come to market that would prove them wrong.

In discussing possible features, most felt that an SDK that provided simple flocking or pathfinding functions might best meet their needs. One developer said he'd like to see some kind of standardized "bot-like" language for AI scripts, though there didn't seem to be any widespread enthusiasm for this idea (probably because of fears it would limit creativity). Also discussed briefly in conjunction with this topic was the matter of what developers would be willing to pay for such an SDK, should a useful one actually be available. Most felt that price was not a particular object; developers today are used to paying (or convincing their bosses to pay) thousands of dollars for toolkits, SDKs, models, and the like. This indicates that if somebody can develop an AI SDK flexible enough to meet the demands of developers, they should be able to pay the rent.

Relic Entertainment's HOMEWORLD used flocking techniques.

Technologies on the Wane

It's become clearer since last year's roundtables that the influence of the more "nontraditional" AI techniques, such as neural networks and genetic algorithms (GAs), is continuing to wane. Whereas in previous years developers had many stories to tell of exploring these and other technologies during their design and development efforts, at this year's sessions there was much more focus on making the more traditional approaches (state machines, rules-based AIs, and so on) work better. The reasons for this varied, but essentially boiled down to variations on the fact that these approaches are better understood and work "well enough." Developers seemed to want to focus much more on how to make them work better and leave exploration of theory to the academic field.

Genetic algorithms have taken a particularly hard hit in the past year. There wasn't a single developer at any of the roundtables that reported using them in any current projects, and most felt that their appeal was overrated. While last year's group had expressed some interest in experimenting with using GAs to help with game tuning, the developers who had tried reported this year that they hadn't found this to be very useful. Nobody could think of much use for GAs outside of the well-known "life simu-

Visibility graphs were used in Red Storm Entertainment's FORCE 21.

Visibility Graphs

One of the interesting areas that game AI is beginning to explore is the realm of terrain analysis. Terrain analysis takes the relatively simple task of pathfinding across a map to its next logical step, which is to get the AI to recognize the strategic and tactical value of various terrain features such as hills, ridges, choke-points, and so on, and incorporate this knowledge into its planning. One tool that offers much promise for dealing with this task is the visibility graph.

Visibility graphs are fairly simple constructs originally developed for the field of robotics motion. They work as follows: Assume you are looking down at a map that has a hill in the center and a pasture with clumps of trees all around it. Let appropriately shaped polygons represent the hill and the trees. The visibility graph for this scene uses the vertices of the polygons for the vertices in the graph, and builds the edges of the graph between the vertices wherever there is a clear (unobstructed) path between the corresponding polygon vertices. The weight of each connecting line equals the distance between the two corresponding polygon vertices. This gives you a simplified map against which you can run a pathfinding algorithm to traverse the map while avoiding the obstacles.

There are some problems with visibility graphs, however. They only give raw connection information, and paths built using them tend to look a little mechanical. Also, the developer needs to do some additional work to prevent all but the smallest units from colliding with polygon (graph) edges as they move, since the path generated from a visibility graph doesn't take into account unit size at all. Still, they're a straightforward way to break down terrain into simplified areas, and they have uses in pathfinding, setting up ambushes (the unobstructed graph edges are natural ambush points), and terrain generation.

lators" such as the CREATURES and PETZ series.

The one exception to this, as previously noted, is the continued use of A-Life techniques. From flocking algorithms that help guide unit formations (FORCE 21, AGE OF KINGS, HOMEWORLD) to object-oriented desire/satisfaction approaches (THE SIMS), developers are finding that these techniques make their games much more lifelike and "predictably unpredictable" than ever before.

Where We're Headed

Always interesting at the roundtables are the inevitable discussions of where the industry in general, and game AI in particular, is headed. As usual, we got almost as many opinions as there were attendees, but some common trends could be seen emerging down the road.

Everybody thought that game AI would continue to be an important part of most games. The recent advances were unlikely to be lost to a new wave of "gee-whiz" 3D graphics engines, and the continued increase in CPU and 3D card capabilities was only going to continue to give AI developers more horsepower. There was the same feeling as last year that the industry would continue to move slowly away from monolithic and rigid rules-based approaches to more purpose-oriented, flexible AIs built using a variety of approaches. It seems safe to assume that extensible AIs will continue to enjoy some popularity and support among developers, mostly in the first-person shooter arena but also in more sophisticated strategy games.

Academia and the defense establishment continue to influence the game AI field (see "Bridging the Gap Between Developers and Researchers," page 34), though it sometimes seems that the academic world learns more from game developers than the other way around. For the most part, developers seem to feel that the academic study of AI is interesting but won't really help them ship their product, while researchers from the academic field find the rapid progress of the game industry enviable even if the techniques aren't all that well documented.

There can be no doubt that the game AI field continues to be one of the most innovative areas of game development. We know what works and tools are beginning to appear to help us do our jobs. With CPU constraints essentially eliminated and the possibilities of good game AI now part of the design process, AI developers can look forward to a bright future of innovation and experimentation.

FOR MORE INFORMATION

WEB SITES

Far and away the best place to find out more about any aspect of game AI is the Internet. There are more excellent web sites filled with tutorials, information, sample code, and so on, than anybody could possibly list in one place. Some of the recommended ones include:

www.gameai.com

Steven Woodcock's site, dedicated to all things game-AI-related. Provides links to other AI resources, reviews on AI implementations in games already on the market, and archives of various Usenet threads.

www.gamasutra.com

The sister site to Game Developer magazine continues to be an excellent discussion area for people with game-AI-related questions. The game AI discussion list there is the largest on the site.

www.gamedev.net

Another excellent site dedicated to all aspects of game development, there is an extensive list of resources and an active discussion group on the topic.

www.red.com/cwr.boids.html

This site remains the single best source for any information about flocking and related A-Life technologies.

www.pcai.com/pcai

PC AI magazine has a marvelous web site crammed with all kinds of useful AI resources. From sample applications to research papers, you can find it here.

http://ai.eecs.umich.edu/people/laird/gamesresearch.html

John E. Laird's site

www.aaai.org

American Association for Artificial Intelligence

NEWSGROUPS

Of course Usenet continues to be a great place to do research on a variety of AI-related topics. The best newsgroups for this purpose remain comp.ai.games, comp.ai, and rec.games.programmers.

PAPERS

Laird, J. E., and M. van Lent. "Interactive Computer Games: Human-Level AI's Killer Application." *Proceedings of the AAAI National Conference on Artificial Intelligence*, August 2000.

Laird, J. E. "It Knows What You're Going to Do: Adding Anticipation to a Quakebot." *Proceedings of the AAAI 2000 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment*, March 2000 (AAAI technical report #SS-00-02).

BOOKS

As Steven Woodcock mentioned in his article, there really aren't too many books that discuss game AI. Probably the best comprehensive reference remains:

Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Upper Saddle River, N. J.: Prentice Hall, 1995.

Bridging the Gap Between Developers & Researchers

One would think that the combined coolness factor of artificial intelligence and computer games would be an irresistible topic, bringing game developers and AI researchers together. Unfortunately, there has been a mutual lack of interest between serious game developers and academic AI researchers. Game developers have picked up a few AI techniques, such as decision trees and the ubiquitous A* algorithm for path planning, but there has been nothing like the knowledge transfer that has taken place with graphics. When game developers look at AI research, they find little work on the problems that interest them, such as nontrivial pathfinding, simple resource management and strategic decision-making, bot control, behavior-scripting languages, and variable levels of skill and personality — all using minimal processing and memory resources. Game developers are looking for example "gems": AI code that they can use or adapt to their specific problems. Unfortunately, most AI research systems are big hunks of code that require a significant investment of time to understand and use effectively.

Why AI Research and Game Development Diverge

AI researchers rarely use computer games for their research, outside of classic board and card games such as chess, checkers, and bridge. Possibly they see most game AI problems as simple "engineering" problems. This view has not been completely unjustified because often the goal of game AI is not to create intelligence, but to improve gameplay through the illusion of intelligent behavior. Many of the techniques used to improve the illusion of intelligence have nothing to do with intelligence, but involve "cheats," such as giving game AIs extra production capability or the ability to see through walls, or "faking it" by creating bots that "talk" to each other but completely ignore what is said. There also has been a drift in AI research toward problems and approaches where precise empirical evaluation is possible. Needless to say, gameplay isn't something that today's AI researchers feel comfortable evaluating.

Although there is currently a significant gap between game developers and AI researchers, that gap is starting to close. The inevitable march of Moore's law is starting to free up significant processing power for AI, especially with the advent of graphics cards that move the graphics processing off the CPU. The added CPU power will make more complex game AI possible. Still, game developers should still be wary of AI researchers who say, "My algorithm doesn't run in real time right now, but just wait. In a few more years, I'm sure the processing power will be there."

A second, equally powerful force that is closing the gap is sociological. Students who grew up loving computer games are getting advanced degrees in AI. This has the dual effect of bringing game research to universities and university research to game companies — already there are at least five AI Ph.D.s at game companies. AI researchers are discovering that building interesting synthetic characters in computer games is much more than just an engineering problem. Moreover, games provide cheap, robust, immersive environments for pursuing many of the core AI issues. They could be the catalyst for a rebirth in research on human-level AI (see my paper on the subject, listed under For More Information, p. 32).

The final force is the game-playing public, who are starting to demand better AI. With the saturation in the quality of computer graphics, better physics and AI are the two technologies that have the most potential to improve gameplay. Players are looking for more realistic AIs to populate their worlds with interesting nonplayer characters (as in THE SIMS) and humanlike opponents who must be out-thought and not just out-shot (and who don't cheat). AI can also provide dynamic game control, adjusting the gameplay based on how the game is played. Imagine playing a first-person shooter where the AI not only reacts to your behavior, but also anticipates your actions by using an internal model of the way you play the game to make its plan. It also adjusts its skill at the tactical level to match yours, so that the game is never a blowout for either side. Our research group has built such a bot using our own Soar AI engine connected to the deathmatch version of QUAKE 2 (see my paper under For More Information). Our research is a peek at what can come out of research labs. The combination of complex AI and computer games can improve existing game genres, and give rise to some new types of games.

Closing the Gap

What can computer game developers do to hasten the collaboration of developers and AI researchers? The most important thing is to make commercial computer game interfaces available to AI researchers. Developers of games such as UNREAL, QUAKE, and HALF-LIFE publish DLLs, making it possible for not only hobbyists but also AI researchers to build bots that play games. If developers from other genres such as real-time strategy games follow suit, you would see an explosion of research on AI for these games. Game developers can also join AI researchers in discussing AI problems and solutions in open forums. There is now a yearly symposium sponsored by the American Association for Artificial Intelligence (AAAI) on AI and interactive entertainment that brings together game developers and AI researchers.

One final note, building good AIs is hard work. Automated learning approaches such as neural nets and genetic algorithms can tune a well-defined set of behavioral parameters, but they are grossly inadequate when it comes to creating synthetic characters with complex behaviors automatically from scratch. There is no magic in AI, except for the magic that emerges when a great programmer works very hard.

AUTHOR'S BIO | *Professor John E. Laird is associate chair of computer science and engineering at the University of Michigan. He is one of the original developers of the Soar AI architecture and leads its continued development and evolution. For the last five years he has taught a senior-level design course on computer game development. He was an organizer of two symposia on AI and computer games and has been a presenter at the last three GDCs.*

The Future Of Game AI

As I slowly reclined back into the seat of the last E3 bus this spring, I was certain of two things: some really great games were coming out in the next year and my feet hurt like hell. A lot of the games that created a buzz featured excellent AI. Since my fellow Ensemble-ites assured me (repeatedly) that no one really cared to hear about my feet, I thought I'd use this space to talk about some of the games coming out in the next 18 months and the new and improved AI technology that will be in them.

Better AI Development Processes and Tools

AI has traditionally been slapped together at the eleventh hour in a product's development cycle. Most programmers know that the really good computer-player (CP) AI has to come at the end because it's darn near impossible to develop CP AI until you know how the game is going to be played. As the use of AI in games has matured, we're starting to see more time and energy spent on developing AI systems that are modular and built in a way that allows them to be tweaked and changed easily as the gameplay changes. This allows the AI development to start sooner, resulting in better AI in the final product. A key component in improving the AI development process is building better tools to go along with the actual AI.

For Ensemble's third real-time strategy (RTS) game, creatively code-named RTS3, we've spent almost a full man-year so far developing a completely new expert system for the CP AI. It's been a lot of work taking the expert system (named, also creatively, XS) from the in-depth requirements discussions with designers to the point where it's ready to pay off. We've finally hit that payoff and have a very robust, extensible scripting language.

The language has been so solid and reusable that, in addition to using it to write the CP AI content, we're using it for console and UI command processing, cinematic control, and the extensive trigger system. We also expect to use XS to write complicated conditional and prerequisite checking for the technology tree; this way, the designers can add off-the-wall prerequisites for research nodes without programmer intervention. Finally, we will also use the XS foundation to write the script code that controls the random map generation for RTS3. The exciting aspect of XS from a tools standpoint is that we will have XS debugging integrated with RTS3's execution. For fans who used the Age of Empires II: The Age of Kings (AoK) expert-system debugging (a display table of 40 or so integer values), this is a huge step up, since XS will significantly increase the ease with which players can create AI personalities.

Better NPC Behavior

In the early days of first-person shooters, non-player characters (NPCs) had the intelligence of nicely rounded rocks. But they've been getting much better lately — look no further than Half-Life's storytelling NPCs and Unreal Tournament's excellent bot AI. The market success of titles such as these has prompted developers to put more effort into AI, so it looks as if smarter NPCs will continue to show up in games.

Grey Matter Studios showed some really impressive technology at E3 with Return to Castle Wolfenstein. When a player throws grenades at Nazi guards, those guards are able to pick up the grenades and throw them back at the player, adding a simple but very effective new wrinkle to NPC interactivity. A neat gameplay mechanic that arises out of this feature is the player's incentive to hold on to grenades long enough so they explode before the guards have a chance to throw them back. Thankfully, Grey Matter thought of this and has already made the guards smart enough not to throw the grenades back if there's no time to do so.

More developers are coupling their AI to their animation/simulation systems to generate characters which move with more realism and accuracy. Irrational did this with System Shock 2 and other developers have done the same for their projects. The developers at Raven are doing similar things with their NPC AI for Star Trek: Elite Force. They created a completely new NPC AI system that's integrated into their Icarus animation system. Elite Force's animations are smoothly integrated into the character behavior, which prevents pops and enables smooth transitions between animations. The result is a significant improvement to the look and feel of the game. I believe that as the use of inverse kinematics in animation increases, games will rely on advanced AI state machines to control and generate even more of the animations. As a side benefit, coupling AI to animation gives you the benefit of more code reuse and memory savings.

Better Communication Using AI

Since the days of Eliza and HAL, people have wanted to talk with their computers. While real-time voice recognition and language processing are still several years off, greater strides are being made to let players better communicate with their computer opponents and allies.

AUTHOR'S BIO | *Dave Pottinger is the technical director at Ensemble Studios. When not working on becoming a Supa-L33t Hax0r, he spends his time with his lovely wife Kristen and their house, Bill. E-mail him at dpottinger@ensemblestudios.com.*

For example, in our upcoming AGE OF EMPIRES: THE AGE OF KINGS expansion pack, THE CONQUERORS, we've enabled a chat communication system that lets you command any computer player simply by sending a chat message or selecting messages from a menu. Combined with AoK's ability to let you script your own CP AI, this lets you craft a computer ally that plays on its own and lets you have conversational exchanges with it in random-map games. This is a small step toward the eventual goal of having players talk to their computer allies in the same way as to humans. Unfortunately, we still have to wait a while for technology to catch up to our desire.

Better Pathfinding AI

In addition to adding great new features, many upcoming games simply have improved on existing AI features, particularly in the area of pathfinding. No one likes screaming at the stupidity of unit movement. Despite the seemingly simple nature of the problem, pathfinding in games has become a big topic in recent years. Many games (including our own AGE OF EMPIRES) have been roasted for bad pathfinding.

In the next year, we will likely see more true 3D games, necessitating the use of pathfinding algorithms that work in three dimensions rather than a hacked-up 2.5 dimensions (two dimensions with a small number of third-dimension planes at fixed heights). Pathing and moving true 3D flying units is much harder than moving units around on the ground, due to the desire to have units bank and turn realistically. So far, no one has proffered a simple solution for pathing in true 3D while taking into account things such as turn radius and other movement restrictions. Instead, most games path without any movement restrictions, use movement restrictions when possible while the unit follows the path, and resort to a contrived turn-in-place approach when movement restrictions conflict with the path.

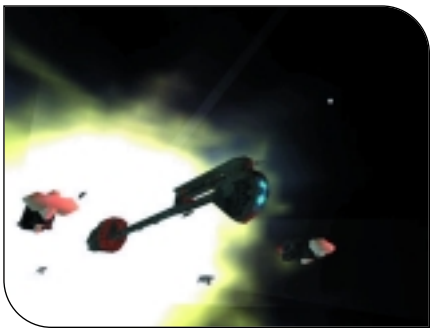
To help compensate for the addition of this extra calculation complexity, we will likely see innovations in the way standard pathfinding algorithms (such as A*) are used. For example, I expect developers will begin to time-slice pathfinding systems so that particularly long routes can be computed over multiple game-world updates and renders. This task can get complicated in a world with dynamic terrain and many moving units, but it can be done if you're willing to spend the memory on it. And improving paths while still maintaining high frame rates is a big advantage.

Also upcoming are more hierarchical pathfinding techniques. Different pathfinding algorithms or data sets can be tuned to a particular need (for example, long or short paths). A hierarchical approach also allows paths to be generated at progressively more detailed levels on an as-needed basis.

Hierarchical AI

Not surprisingly, RTS games have some of the most demanding AI needs. Their AI has to meet a player's expectations of a challenging strategy game, yet still make decisions within milliseconds in order to meet the game's frame-rate requirements. Hierarchical approaches to AI have been successful in helping address these needs.

In hierarchical RTS AI, there are differ-



TOP. The CONQUERORS expansion pack builds on the AI engine from AGE OF EMPIRES II: THE AGE OF KINGS.
MIDDLE. ELITE FORCE showcases an excellent combined animation/AI system.
BOTTOM. Interesting new entity AI features will be a key component in CATACLYSM.

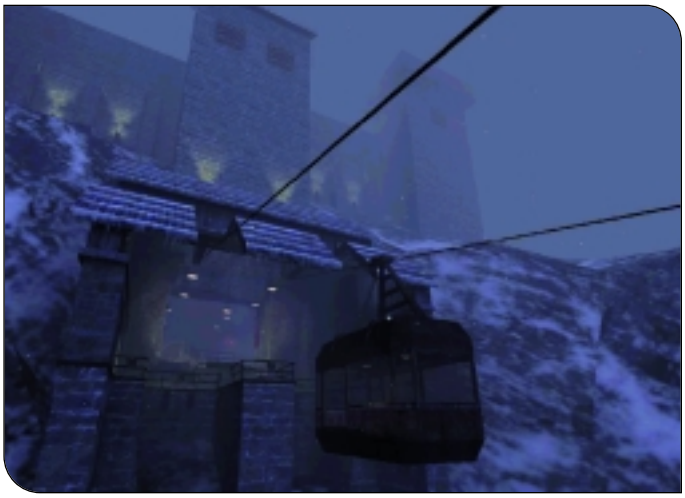
ent layers to the AI. The strategic AI makes high-level decisions such as "What units should I train?" The tactical AI executes the orders given by the strategic AI in the best possible way, deciding things such as where to train the units requested by the strategic AI. Usually, the strategic AI is evaluated far less frequently than the tactical AI. There's often a third layer, which we'll call entity AI. Entity AI represents the physical entities in the game, such as units or groups, and is manipulated by the tactical AI. Thus, the entity AI is usually processed more frequently than the tactical AI (particularly if the entity AI has combined AI and animation responsibility).

As the genre matures, RTS developers are finding more interesting ways to use this type of system. The upcoming HOMEWORLD sequel, CATACLYSM, builds heavily on the idea of combining simple AI behaviors. Unit aggression stances are used typically to control how far units pursue enemies. That concept is combined with the simple idea of patrolling between two waypoints. So, if the units are set in an aggressive stance while patrolling, they will attack any targets they come across. However, if the units are set in an evasive stance, they will avoid enemy contact during patrols. While this isn't hard to do (assuming the code is written well), it's an example of how the entity AI can evolve to become more complex.

THE CONQUERORS features another example of behavior combination. In AoK, your villagers stand around loafing after finishing that lumber camp on the edge of your town. In THE CONQUERORS, villagers are smarter; they begin chopping wood after finishing constructing a lumber camp. Again, this seems simple, but it makes for a much better game (and was one of the most well-received features by AoK fans at E3). Features such as this can also help offload responsibility from the oft-overburdened tactical AI. If the tactical AI can rely on villagers to keep working after building a resource drop-site, it can remove another round of villager-tasking from its plate.

A little farther out on the RTS horizon are our own RTS3 and Blizzard's WARCRAFT 3. Both will rely heavily on autonomous agent behavior (a fancy name for entity AI). Similar to combining simple behaviors, an event-driven hierarchical entity AI can alleviate a lot of needless AI polling by executing code only when there is a reason to do something. This frees up processor time for more AI, graphics, and other tasks.

A comprehensive group-AI system also makes it a lot easier to implement features such as group-based protection. Imagine that you've ordered a group of melee units to protect some ranged units. If the ranged units aren't in danger or actively taking damage, you probably want the melee units to go beat on something. However, as soon as the ranged units take damage, you want the guarding melee units to rush over and stomp the attacking units. This is possible in a non-group-AI system, but it requires very clunky data structures and is a lot harder to achieve. And if it's a lot harder to code, then it will take longer to develop and be less robust (read: really, really buggy). On the other hand, if you have a group system you can simply pass the damage notification up to the group and let it quickly iterate through its guarding units, commanding them to attack the evil enemy units as necessary.



TOP. Watch out for those tricky grenade-throwing guards when you get off of the gondola in RETURN TO CASTLE WOLFENSTEIN.
BOTTOM. UNREAL TOURNAMENT delivered on the promise of excellent bot AI.

Fun versus Difficulty

One long-standing AI question is, "To cheat or not to cheat?" It used to be that game developers had to bypass the game rules that bound players in order to empower the AI. Weak AI can hurt the game experience, and allowing the computer to cheat was the only way around that problem. Happily, that's been changing over the last few years. Several games have been released in which the AI has at least some difficulty levels that don't cheat (AGE OF EMPIRES, for example). This has all been done under the assumption that increased difficulty means more fun. A better, more difficult AI is more fun to play against, right? Not always.

Fresh from of a frustration-filled game against a few of THE CONQUERORS' AI opponents, Tim Deen (one of Ensemble's designers) sent out an e-mail declaring that he really wished we'd focus on making the AI more fun to play against for the RTS3 project. Some healthy discussion ensued and we discussed the relative complexity of making an AI player harder to play against versus more fun. The consensus was that it was a lot easier to make an AI more difficult to play against. So, being good lazy programmers, we had done just that without really giving it much thought.

As we start to build AI systems that can stomp good players into the ground fair and square, we need to look at the next step. That next step should be making the

game fun. Since it's not much fun to play against an AI that never has a chance to beat you, the AI has to be able to put up a really good fight. Naturally, we have tools to do that, and it's easy to measure the success of that approach using lots of fun spreadsheets and graphs. It's more difficult — and, more significantly, considerably more subjective — to make an AI fun to play against. Conveniently, many of the tools that we already have from building difficult AIs can be leveraged to make the game more fun to play.

UNREAL TOURNAMENT has some great bot code that can really compete with the best players. Yet, it's also fun to play against. It intentionally makes mistakes and doesn't always do the best thing it can. While that may not be the most interesting thing from an academic AI perspective, it's a lot more fun than getting shot in the back every single time.

In our RTS3 project, we're going to use the XS scripting language to control the level of difficulty. Since we have an idea of how long we'd like each game to take, our AI designers can check things such as game time, how many of the CP's units have been killed, how many of the human player's units were killed by the CP, or the score of the game to see who's "winning." Armed with that information, they can scale back the quality of the AI to make sure the game doesn't drag out long after the outcome is really determined. If you start to augment that ability with other features such as game history logging, you have the makings of a good opponent that quickly scales to your initial difficulty level and continues to give you a challenging game even as you get better.

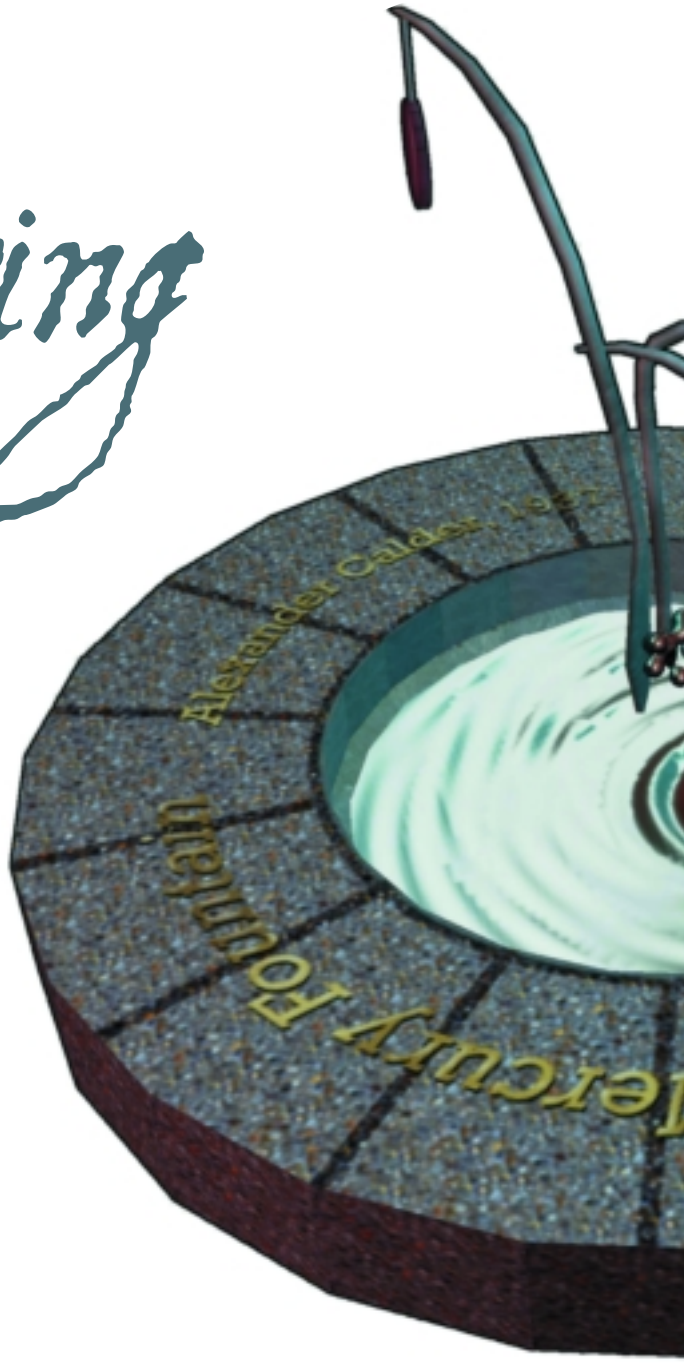
WARCRAFT 3 sports a complete overhaul of everything in the engine, including the AI.

The next year or so still looks to have a heavy focus on graphics, particularly as X-Box, Playstation 2, Dolphin, and the good old PC continue to vie for visual supremacy. But, perhaps less glamorously, AI keeps chugging along and is getting better. As more developers dig into AI and realize that good AI is just as difficult as pushing tons of polygons to the screen, AI is getting increased attention. Almost every developer at E3 had an answer to the question, "So what new AI stuff are you doing with this game?" You can't get much cooler than that, which is why I'm optimistic about the continuing improvement and refinement of AI in games.

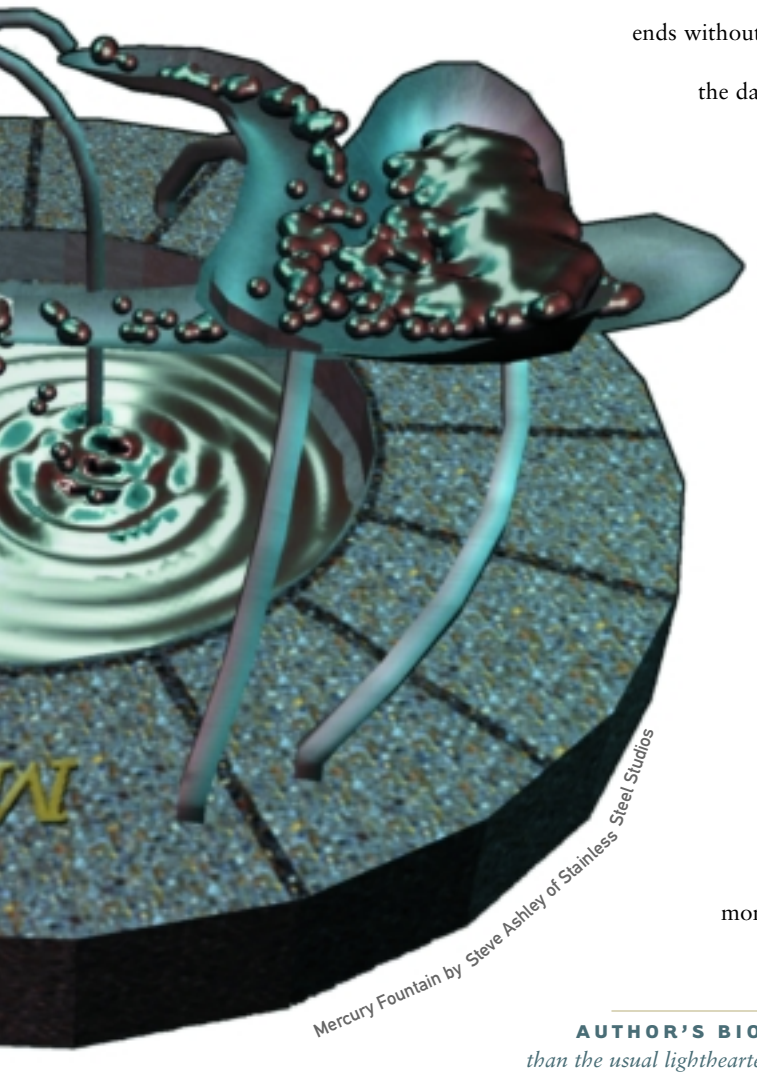
brian sharp

Moving Fluid:

The Conclusion of a Two-part Series about Implicit Surfaces



A two-article series is the ultimate in delayed gratification. The first month you have to sit there and read a bunch of math, and you get nothing for your work save a few measly screenshots. Even worse, it ends without tying up any loose ends, and just leaves you sitting there in the dark.



Mercury Fountain by Steve Ashley of Stainless Steel Studios

Then you have to wait an entire month before you get to read the second article, which brings everything together and gives you a juicy demo to sink your teeth into. Well, good news: since this is the second article, I'll (hopefully) provide some closure and I'll definitely provide a demo.

For any of you who skipped class last month ("Go with the Flow: Improving Fluid Rendering Using Implicit Surfaces," July 2000), this article probably won't make very much sense. Having read last month's article is a pretty firm prerequisite to making it through this month's material, so grab last month's issue and chow down.

AUTHOR'S BIO | *I, Brian Sharp, can be reached at brian@maniacal.org. Rather than the usual lighthearted bio this month, though, I'm taking this space to dedicate this series of two articles to a very good friend who died in March 2000. Seumas McNally of Longbow Digital Arts was an inspiration and great help on these two articles, as for all my others, and I'll think back fondly on our conversations even if he never did like martinis. Seumas, farewell. We'll all miss you.*

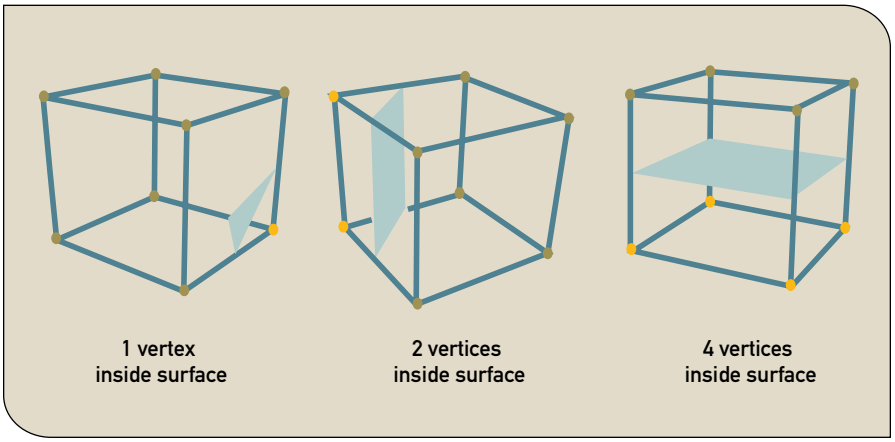


FIGURE 1. Example cubelet configurations and resulting polygonizations. Orange vertices are inside the surface, green are outside.

Now, Where Were We?

So, in case you've forgotten, we're trying to model liquids using implicit surfaces. Last month I covered quite a bit of ground. I described the technique for modeling surfaces using a number of molecules. I then defined the value of our implicit function: Let p be the point in space for which you want to evaluate the implicit function. Then let c_i be the location of the ith molecule (where the total number of molecules is n). The value of the function at p is then:

$$\text{potential}(p) = \sum_{i=0}^{n} \max\left(0, \frac{1}{|c_i - p|^2} - 1\right)$$

From there, I examined the basic marching cubes algorithm used to tessellate the surface. I broke space up into a uniform 3D grid and for each grid cell sampled the function and generated polygons based on the results. You can see some example tessellations in Figure 1.

Slow Going at First

Using the basic, brute-force marching cubes algorithm, you can already generate a full surface: vertices, normals, texture coordinates, colors, the whole deal. So what do we have left to do? Well, as I pointed out last month, the brute-force marching cubes algorithm is slow. Very slow.

The first thing I'll cover in this article is a brief (and informal) overview of algorithm analysis. After all, we can't very well optimize our algorithms if we don't understand why they're slow in the first place. After that, I'll detail the process of optimizing brute-force marching cubes with the goal of creating a process that runs in real time and scales well with larger surfaces.

Finally, after that, I'll discuss physics. It's one thing to be able to render an implicit surface, but it's another thing to make that implicit surface move like fluid. We'll look at methods for modeling fluid cohesion and flow, incorporating gravity and other external forces, and dealing with collisions against the rest of the world.

Algorithms in Brief

If you already have a grasp of algorithm analysis, feel free to skip this section, but understanding running times is an important prerequisite to getting our fluid rendering to run at a good speed. Also note that these few paragraphs provide just the bare minimum necessary to understand the article, and are no substitute for a good reference on algorithms. My personal favorite algorithm text is *Introduction to Algorithms*, listed in the For More Information section at the end of the article.

Any algorithm, such as a sorting algorithm for example, has what's known as a running time. There are different ways to refer to running time, but the notation we care about is what's known as "big-O" notation. You may have seen it before, something to the effect of, "This algorithm runs in $O(n^2)$ time."

Big-O is an upper bound on the running time. It means that the algorithm takes at most n^2 time. The n^2 is a reference to how the algorithm scales. The n refers to some variable. For a sorting algorithm it's almost always the number of items being sorted.

In this example, if the sorting algorithm was $O(n^2)$, it means that doubling n — the number of items being sorted — will quadruple the time it takes to sort them: sorting 20 items will take four times as long as sorting 10 items, since $20^2 = 4 * 10^2$. To make an analogy to code, n^2 is just like a doubly-nested `for` loop, where for every element in an array, it walks through and considers every other element in the array.

Other common running times include $O(n)$, known as linear time, since the execution time scales linearly with the input. Also, $O(\log n)$ is not uncommon. Logarithmic time usually has something to do with walking a tree data structure. If we're recursing down an octree for instance with n leaf nodes, we have to walk down $\log_8 n$ nodes to get from the top to a leaf node.

Running times are also subject to common arithmetic: if we have an algorithm that runs in $O(n)$ time and at every step of the way it executes another algorithm that runs in $O(n)$ time, the total algorithm takes $O(n) * O(n) = O(n^2)$ time.

One caveat to all this running time stuff is that it obviously says nothing about how long it actually takes to run the algorithm, which can be in milliseconds, seconds, or years. The time depends on a number of things, other constants such as the speed of the machine, and how many machine instructions the algorithm compiles into. But that's not to say the constant is unimportant: if it took a year to tessellate a really basic droplet of fluid, all the scalability in the world would be pointless, because the algorithm would simply run too slowly.

Luckily, that's not the case. Even the brute-force marching cubes algorithm can tessellate a small surface quickly, which means our constants are O.K. So the question is, what's wrong with our scalability?

Optimize!

To answer that, we need to figure out the running time of our brute-force marching cubes algorithm. This process isn't quite as easy as the one for a sorting

algorithm because it's not immediately clear what our n should be — what's the variable we're measuring?

Consider the bounding box of the implicit surface. Since the number of cubelets sampled is directly related to that bounding box's volume, we'd like to relate the running time to that volume. So, I'll choose n to be the length of one of the edges of the bounding box. Here I'm assuming that the bounding box is roughly cubical. Given that, the brute-force marching cubes algorithm is about $O(n^3)$, since the number of cubelets it considers is relative to the box's volume, which is about n^3.

Suffice it to say that we'd like to be able to do better than n^3. And, intuitively, we should be able to. Consider that the brute-force marching cubes algorithm is sampling the entire 3D volume. But the surface itself is really only two-dimensional: it bounds a 3D volume, but it — the boundary — has no volume. I'll admit to a lot of hand-waving here, but practically speaking, as we crank the tessellation density up, the surface will occupy a very small percentage of the cubelets in the bounding box.

Therefore, if brute-force sampling of the entire volume gives us $O(n^3)$, being more efficient should hopefully let us get closer to $O(n^2)$, since we're stepping down from sampling a 3D volume to, ideally, a 2D surface. If we could get close to that, it would make a huge improvement in our running time.

Sounds Good, but How?

The general idea for achieving this increase in speed is pretty obvious: the vast majority of the cubelets currently being sampled are empty, so we should stop sampling all those empty ones. What's not necessarily obvious is how we do so. There are two techniques commonly used: the octree method and the surface crawler.

The octree method. The first technique uses an octree to partition the surface's bounding box. Instead of treating space as a 3D grid, we treat it as an octree with those cubelets as its leaves. In order to tessellate, then, we don't just iterate through all the cubelets. We start at the top of the octree and if any of that node's child nodes contain the surface, we recurse on them. When we get to leaf nodes, if they contain the surface, we polygonize them with the standard marching cubes algorithm. A tessellation done with this technique is shown in Figure 2.

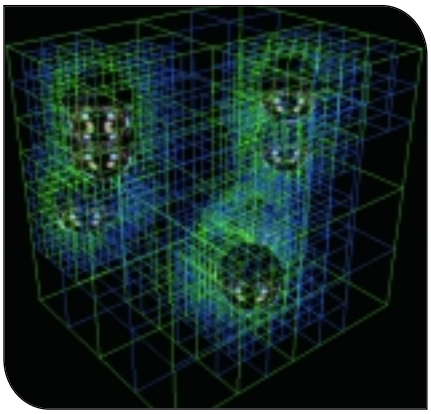


FIGURE 2. An example of octree tessellation: blue boxes are cells considered but empty, green boxes are cells considered that contain the surface.

Now, to analyze the speed-up. As the algorithm recurses through the octree, it will eventually touch every cubelet on the surface — and there are $O(n^2)$ of them. It will also hit every node in the octree on the way down to those $O(n^2)$ cubelets, and if you add all those up, they're on the order $O(n^2)$ as well. This leaves our algorithm running at our goal of $O(n^2)$ total.

Does this mean we're done? Well, let's look at some of the downfalls of this algorithm. The first is that it's very hard to determine at each step down the octree whether a node contains the surface or not. Remember that for the brute-force marching cubes algorithm we decided that we didn't care whether we undersampled the surface or not; the solution was just to crank up the tessellation. Unfortunately, with the octree approach, we can't afford to undersample.

Consider a larger node higher up in the octree that contains a large blob of fluid, but the blob doesn't touch any of the edges of the node's bounding box. If we don't notice this and just check the corners of the node's box to determine whether we should recurse, we'll miss this entire blob of fluid. That's basically unacceptable and no matter how fine the final tessellation is, the higher nodes will have large bounding boxes. As a result, the act of determining whether to recurse on a node is quite complicated — we need to consider whether any molecules are in or near the box, which amounts to a lot of work.

Furthermore, when I implemented this scheme to see how it worked in practice, I found that it still does a lot of unnecessary work. For example, when the algorithm reaches a second-to-bottom node in the octree, it has eight child nodes to consider, and it'll execute the standard marching cubes evaluation on each one. However, it could be that only a single one of those cubelets actually contains the surface. You can see this a bit in Figure 2, where it spends time considering many cubelets near, but not on, the surface. Therefore, it's worth looking at the second speed-enhancing technique to see if it's any better.

The surface crawler. This technique, the surface crawler approach, takes advantage of the fact that the surface is continuous — that it has no open holes. The algorithm first finds a cubelet on the surface, polygonizes it, and "crawls" to all neighboring cubelets that also contain the surface. It recurses until it has polygonized the entire surface.

Let's analyze the speed-up. The algorithm takes some amount of time to find that initial cubelet on the surface. Furthermore, if the surface is broken into multiple pieces, it takes some time for each surface to find an initial cubelet. But this work is dwarfed by the work it does recursing over the surface. And since it considers every cubelet on the surface exactly once, this algorithm too runs in $O(n^2)$ time.

One of the detriments of the surface crawler technique is that it has some tricky details. When it traces out the surface, it forms what's known as a directed cyclic graph. That is, the path it walks over the surface potentially has cycles — if you just blindly recurse, you'll end up in an infinite loop. Thus, be careful when recursing to a cubelet's neighbors so you only recurse to those that haven't previously been polygonized. This requires the use of a hash table, and that adds a hash table lookup per cubelet.

On the other hand, a benefit of the surface crawler technique is that it doesn't consider a bunch of cubelets that aren't on the surface; other than initially finding the cubelet on the surface, every cubelet it considers is guaranteed to contain the surface.

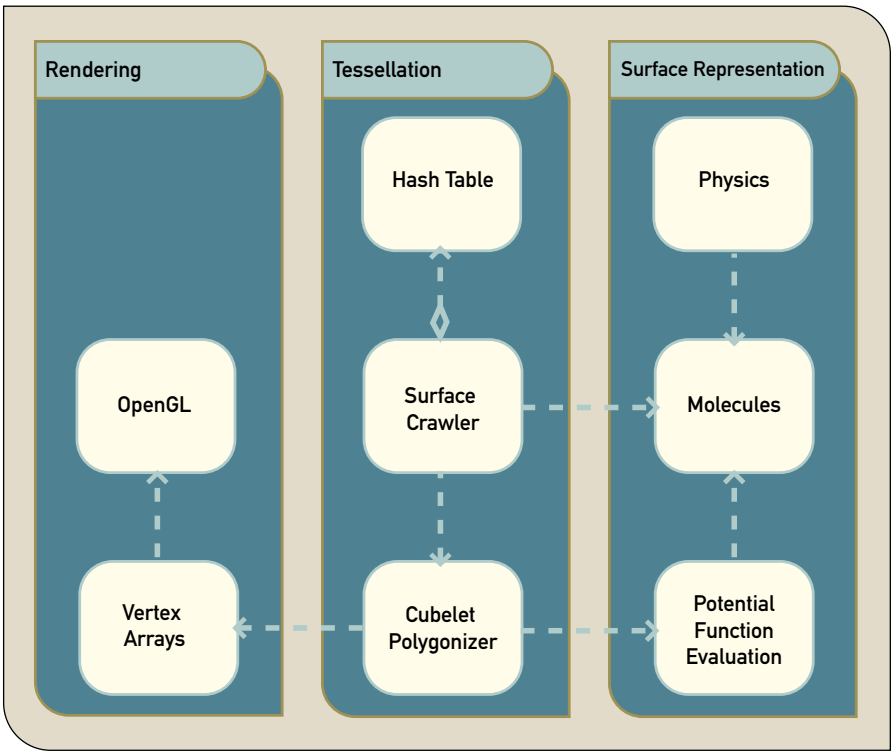


FIGURE 3. A high-level UML diagram of the entire fluid-flow system.

Deciding between this technique and the octree one comes down to a question of constants — they both scale the same way but one is likely slower by a constant factor. In this case, I found the surface crawler algorithm generally faster and so it's the one I've used for the sample implementation.

Surface crawling specifics. I left out some implementation details while trying to decide between octrees and surface crawling; now that I've chosen my method it's time to get into specifics. The first process that I didn't specify in much detail was how we find the initial cubelet on the surface to start recursing from. To do this, we start a cubelet at each molecule's location and march it outwards until it lands on the surface. This usually takes only a few steps for each molecule; once the cubelet reaches the surface, we recurse to find the rest.

Determining which neighboring cubelets contain the surface is easy, as it falls right out of the polygonization. Consider Figure 1 again. Note the third case with the polygon right through the middle of the cubelet — it's obvious that all its horizontal neighbors will contain the surface. In general, for any cubelet face with a polygon edge across it, the neighboring cubelet that shares that face will also contain the surface.

Figure 3 shows a UML diagram of a piece of our implementation. The surface crawler first uses the molecules to find the initial cubelet on the surface. Then the surface crawler uses the cube polygonizer to assemble the polygons for each cubelet, and also to return neighbor information. The cube polygonizer uses the potential function evaluator as part of polygonization, and it deposits the generated polygons in the vertex arrays.

The surface crawler stores a hash table that maps cubelet locations to a flag. If the flag is set, it means that you've already considered that cubelet. To avoid looping infinitely, every time you polygonize a cubelet, set its flag in the hash table, and when recursing to neighbors, don't recurse to those whose flags are set.

One particular nitty-gritty detail of optimization that matters quite a bit is the choice of hash function. It must be fast to evaluate, since you'll be evaluating it for every cubelet on the surface. But you also need to make sure that it produces a uniform distribution or else the hash table distribution will be poor and accesses will be slow. What I do is store each cubelet's address as three shorts, an (x,y,z) triple. Then, to generate a hash key, I just chop off the higher-order bits of each and concatenate them. This makes the hash function act like a mod table in space, where nearby cubelets hash to different hash buckets. In practice, this generates very even distributions.

Still Not Fast Enough!

There's actually a variable that I've been leaving out of the running times that matters a whole lot, which is the number of molecules. For instance, I mentioned previously that the running time of the surface crawler is roughly $O(n^2)$. But really, for each cubelet that's evaluated you have to iterate over all the molecules to evaluate the implicit function. That means that each cubelet polygonization is $O(m)$, where m is the number of molecules. Taking this new variable into consideration, the surface crawler is actually $O(mn^2)$.

This is a pretty big deal, too, considering that the surface may have many hundreds of molecules in it, only a very few of which will be near enough to any given point to contribute to its potential. Recall that our falloff function is designed to hit 0 at a distance of 1.0 from the molecule, for the specific purpose of limiting the number of molecules that could affect a point's potential.

Furthermore, because we're modeling liquid and liquid is noncompressible, the physics system should always keep the molecules from packing tightly together. This further ensures that you'll never have all the molecules contributing to a single point's potential.

So, what to do about this? Here I introduce yet another spatial partitioning mechanism to the system. Don't worry, though, it's pretty straightforward. I just break the surface's bounding box into a uniform set of grid cells — sound familiar? — and sort the molecules into their respective cells. The difference between this grid and the marching cubes grid, however, is that this grid is much coarser: each grid cell is large enough to contain a molecule's entire influence sphere. This way, whenever you have a point in space for which you'd like to

evaluate the potential, you only need to consider molecules in that point's cell and the surrounding 26 cells, since anything farther away is guaranteed to be more than 1.0 unit away.

Maintaining the molecules in their cells is simple, too. Each cell is just a pointer, and to add molecules to it you chain them on the end of the pointer, forming a linked list. For every frame, you clear all the cells and rebuild the lists from scratch.

With this system in place, there's a fixed maximum number of molecules that you'll ever consider in evaluating the potential function at a point. This is great, because it brings the running time for the surface crawler back to $O(n^2)$, since the number of molecules in the entire surface has no bearing on how many are near a particular point.

There is a downside to this approach, though: as the bounding box of the surface grows, the amount of allocated memory grows to $O(n^3)$ since we blindly allocate empty cells regardless of whether they will contain molecules or not. This means that if the surface is spreading out a lot, or even if a single molecule goes flying into the air or something, the system can grind to a halt as it allocates a huge number of empty cells.

This hasn't been a problem so far in my implementation, but if it should prove a problem, there's a solution that trades off memory for a little bit of running time. Instead of storing the molecule cell table as a 3D array of grid cells, we could store it as an octree, only allocating lower nodes when a molecule needs to be added to them. Sounding familiar again? Indeed, this is a lot like the octree approach to tessellation, and where that achieved some running time improvements, this wins some space improvements: the total amount of space needed comes down to $O(m)$ since you'll never need more cells than there are molecules.

The downside to this, though, is that looking up a molecule becomes more expensive, since you can't directly index into a grid cell as you can for the 3D array. It requires traversing down the octree, which is $O(\log n)$, since the depth of the tree will still depend upon the surface's bounding box. This would bring our total running time for the surface crawler to $O(n^2 \log n)$, which is less desirable than $O(n^2)$. For some applications, though, it might be necessary to conserve memory.

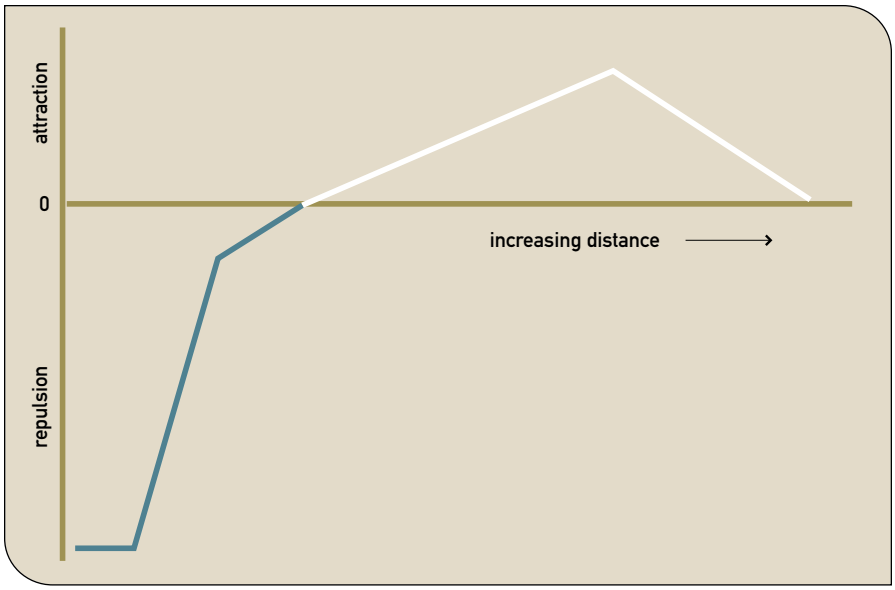


FIGURE 4. The force function used for molecule vs. molecule interactions.

Phew! Fast Enough?

Well, we've succeeded. The original running time of the brute-force marching cubes algorithm was a fairly atrocious $O(mn^3)$. Through the surface crawler and the spatial partitioning of the molecules, it's now down to $O(n^2)$, which is a tremendous speed-up. Again, these running times are approximate, and there are pathologically bad cases (very complex surfaces) that will still be slow, but in practice the algorithm is now fast enough for real-time use. Now, if only we had something real-time to do with it!

Physics. Far be it from me to rehash information that's already been covered in *Game Developer*. If you haven't read Chris Hecker's four-article series on rigid-body dynamics (Behind the Screen, Oct./Nov. 1996–Feb./Mar. 1997 and June 1997) or Jeff Lander's forays into physics ("Physics on the Back of a Cocktail Napkin," Graphic Content, September 1999), you should do yourself a favor and read them.

However, our physics system has some unique characteristics that demand choices different from those you might make for, say, a racing game. For instance, in a racing game there are maybe 16 cars. At that level, you can afford to give each of them a detailed physics model and deal with collisions in expensive, analytically correct ways.

For our application, however, there are hundreds of molecules. Each of them is always moving and colliding with things, and to treat each one with the accuracy of a race car model would bring the system to its knees. The most important thing to keep in mind is that the physical model for the molecules needs to be kept simple or else you'll pay for it later.

On our side, though, are some nice simplifications you get for free. The most notable of these is that, as particles, the molecules don't rotate or spin, so you don't have to deal with torque. Furthermore, you can ignore friction since it doesn't make sense — at least, not in a simple enough way — to talk about friction of water against a surface.

So all you really need to worry about are two things. One is the molecular interactions: how the molecules exert forces upon each other to give the appearance of cohesion and flow. The other is how molecules collide with things and how you respond to those collisions.

Intermolecule Forces

In order to make fluid look convincing and not like a bunch of points moving around independently, you need some

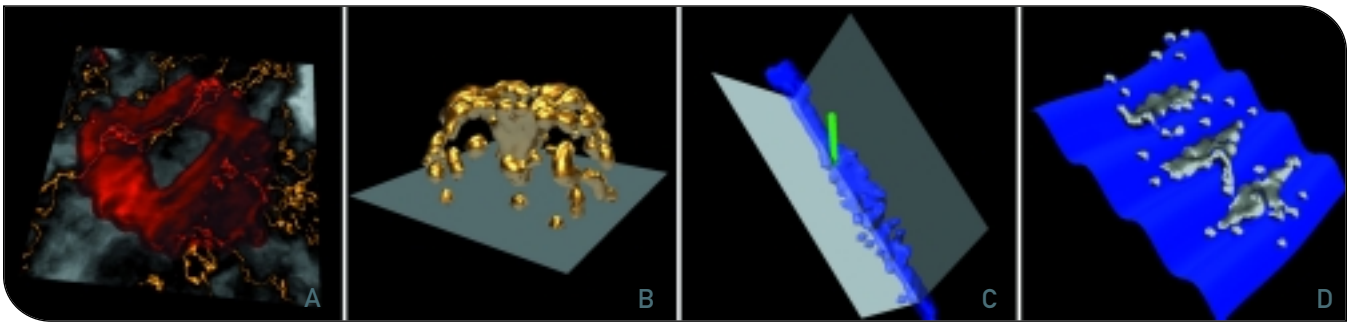


FIGURE 5. Screenshots from the demo. (A) Fluid cohesion. (B) External forces — gravity. (C) Collision against planes and cylinders. (D) Collision against height fields.

forces to make it look cohesive but also to keep the points from piling together.

The forces I use are shown in Figure 4. At very close distances, I consider the molecules to be interpenetrating and hit them with a strong force to prevent interpenetration. Somewhat farther away, I repel them to an equilibrium point. If they move farther away than that, I exert an attractive force to keep them together, which eventually falls off to zero: the fluid is only attractive locally.

Furthermore, I supplement these forces using damping forces based on velocity: if two molecules are in the collision zone and are still moving together very quickly, I exert more repulsive force. If they're moving apart, I exert less. Conversely, if they're in the attraction zone and moving together, I exert less attractive force, if they're moving apart, I exert more.

World collisions. Collisions, collision response, and resting contact are easily the most complex and involved aspect of physical simulation. Generally speaking, we have two routes to go: the analytical route, or the "looks good enough" route. The analytical route has some issues with worst-case running times and unluckily for us, we tend to hit those fairly frequently: when all the molecules pile on top of each other, finding the forces to prevent interpenetration skyrockets to intractability.

So I chose to go with somewhat more forgiving alternative penalty methods. The basic idea is that you do collision resolution by exerting a spring force and damping force on colliding objects. The spring force increases as the objects interpenetrate more in the hopes of preventing interpenetration. The damping force is based on the objects' velocities: if they are moving into each other, the damping force augments the spring force; if they're moving apart, it damps the spring force. This doesn't guarantee that things won't interpenetrate — in fact, it virtually ensures that they will a little bit. But it provides some wiggle room, and for our purposes it's O.K. if things aren't perfect as long as they look good.

The payoff is that the simulator can then run in O(m) time even under the worst conditions. A molecule doesn't need to know about any molecules other than the ones it's colliding with, and we never end up solving for global solutions. The other nice thing about penalties is that they keep the entire physics model force-based. To the simulator, the penalty repulsive forces look exactly like any other force, such as gravity or wind.

The downside to penalty methods is that the spring forces can get very large, or "stiff," and if we're not careful things can explode all over the place. Keeping things stable is the job of our integrator.

Integration. I'm going to assume you're familiar with the concepts behind numerical integration or else you're going to have a tough time working with computational physics. Chris Hecker has some great references on his web page (www.d6.com/users/checker/dynamics.htm) if you feel the need for a refresher.

While integrator stability is always important, it's even more important than usual here. This is another side effect of the large numbers of molecules. Say that a single molecule slams into the ground very hard, hard enough to generate a penalty force that makes the integrator blow up. If we're unlucky, this can start a chain reaction by jostling nearby molecules around — again, hard enough to cause some problems.

It's a little hypocritical of me to write about how important stability is, because as of this writing I'm using a brain-dead explicit Euler integrator — the lowest of the low — which is neither particularly accurate nor at all stable. In some of the demos, I ended up cranking the physics frame rate up to 100Hz just to keep things from blowing up.

However, there is a solution. It's just not particularly easy. If instead of an explicit Euler I used an implicit Euler integrator, sometimes called a "backwards" Euler integrator, most of my instability problems would go away. The downside is that implicit integration is more computationally expensive, a downside that would probably be offset by being able to run the physics at a much larger timestep.

On with the Demo

Your reward for reading all the way through these two articles (or at least this one) is a fairly comprehensive demo. Ceding my programmer-caliber artistic skills, I enlisted the help of Steve Ashley, currently working at Stainless Steel Studios, for the model of the fountain. For a little historical background, the model is of the Mercury Fountain by American artist Alexander Calder. Calder originally designed the Mercury Fountain for the Spanish Republican pavilion at the Paris Universal Exhibition of 1937. The piece honored the mercury-mining town of Alamdén, a region that was devastated during the Spanish Civil War. Calder

donated the Mercury Fountain in 1975 to the Fundació Joan Miró in Barcelona where it resides today (see the Fundació Joan Miró web site, in the For More Information section).

If you can't get to Barcelona, though, this demo is the next best thing. That fact that the fountain runs with mercury is perfect: mercury is much easier to render than a transparent liquid. For mercury, a good shiny environment map will do, whereas for transparent liquids it's hard to mimic effects such as refraction realistically.

Along with the fountain, the demo includes a number of other simulations of more focused effects — molecular attraction and repulsion (Figure 5a), external force exertion (Figure 5b), collision with basic primitives (Figure 5c) and collision with height fields (Figure 5d).

So Now What?

The obvious next step is to take this technique and put it in your game today. I want to be clear about this: I want better water in games, better lava, better liquid in general, and I want it now. All joking aside, though, I don't want to imply that you could actually drop this into your engine in an hour and a half; there are admittedly some obstacles with this technique. First of all, the tessellation and rendering are not fast enough to render, say, an entire ocean. Furthermore, the physics computation can take quite a bit of time with involved surfaces on complex height fields.

That's not to say the technique doesn't have some promising uses. There are a number of possible ways to add dynamic fluid to your game. The first possibility is to use it as "window dressing" — have the faucets in the gritty subway bathroom of your first-person shooter be broken and spraying on the floor. Or use it for some spell effects in your role-playing game. Something like that.

What I'd prefer, though, is a use that makes this technique an important part of the gameplay. It's possible, it just has to be done carefully. For example, in the example I mentioned last month of diverting a river to flood an enemy base, you probably wouldn't want to render the entire river with implicits. Instead, use the jiggly plane technique wherever the river is steadily running, and only use implicits at the head of the river as it rushes down its new path. Some distance back from the head, stitch the implicit tessellation to a jiggly plane, and you're set.

One way or another, I hope these two articles have been useful and I hope I'm not the only developer thinking about ways to improve our fluid rendering techniques. While you're thinking of the perfect way to integrate this into your upcoming hit title, enjoy the demo, and feel free to e-mail me (please!) with any thoughts, questions, or comments on these articles or the demo.

FOR MORE INFORMATION

WEB SITES

Author's site
www.maniacal.org

Chris Hecker's dynamics page
www.d6.com/users/checker/dynamics.htm

Fundació Joan Miró
www.bcn.fjmiro.es

Zack Simpson's Virtual Calder
www.totempole.net/balance/balance.html

ArgoUML Open-Source CASE Tool
www.argouml.org

BOOKS

Cormen, T., C. Leiserson, and R. Rivest. *Introduction to Algorithms*. Cambridge, Mass.: M.I.T. Press, 1998.

Bloomenthal, Jules, and others. *Introduction to Implicit Surfaces*. San Francisco, Calif.: Morgan-Kaufmann, 1997.

ACKNOWLEDGEMENTS

Thanks to Kent Quirk of CogniToy for discussions, proofreading, and camaraderie while I was writing this and past articles. Thanks to Steve Ashley of Stainless Steel Studios for the model of the Mercury Fountain. And thanks to Zack Simpson for reminding me of my interest in Alexander Calder with his Virtual Calder simulation.

Psygnosis's Colony Wars: Red Sun

GAME DATA

PUBLISHER: Sony Computer Entertainment Europe/America
FULL-TIME DEVELOPERS: Five full-time programmers, five artists, three designers, one musician
CONTRACTORS: Video — Havoc
BUDGET: $2.35 million
LENGTH OF DEVELOPMENT: 14 months
RELEASE DATE: Spring 2000
PLATFORM: Playstation
HARDWARE USED: 600MHz Pentium IIIs with 128MB RAM, SN Systems' Playstation dev kit
SOFTWARE USED: Softimage 3.8, Microsoft Visual Studio 6, Microsoft Visual Sourcesafe 5, Gnu C++, Watcom WMAKE
LINES OF CODE: 220,000

The Colony Wars: Red Sun development team.

When the Sony/Psygnosis Leeds studio opened in 1996, it became the Liverpool-based company's sixth development house. Although the studio's initial titles — which by most standards were good products — didn't achieve particularly impressive sales or market penetration, the studio had proved itself to be capable and professional enough to be entrusted with two of the company's key intellectual properties: Wipeout and Colony Wars.

I joined the Leeds studio in late 1998. At that point, the Colony Wars series, developed by our sister studio in Liverpool, had seen two incarnations, both of which had won critical acclaim. They became the definitive 3D space combat games, with fluid controls, epic full-motion video sequences, and flashy special effects. Now the torch was passed on to us, a completely new team in a different studio, to carry on the good work with Colony Wars: Red Sun.

Inauspicious Beginnings

The initial briefing from our Liverpool studio's marketing department was not encouraging. Colony Wars: Red Sun was intended to be a bridge title, a smooth path between Colony Wars 2 and, potentially, Colony Wars 4 on the Playstation 2 (which would be developed sometime in the future). Consequently, we faced a very short development schedule — about six months of full development before we had to complete our alpha.

Our development process got off to a shaky start. Most of the production staff and experienced Playstation programmers in the studio were still tied up with ongoing projects, so it fell to Gareth Preece (who had a small amount of Playstation programming experience to his credit) and me (a die-hard PC programmer) to start the programming ball rolling.

The art team was less pressed for resources. We had five experienced artists from day one: Chris Hogg, Pete Owen-James, Roger Coe, Andy Hanson, and Ben Devereau. They came up with concepts on paper, and dissected the models and extraction tools from the previous games to see how the underlying systems ticked.

On the design side, Paul Walker, Simon Stratford, and Jody Cobb were starting to piece together some initial ideas for enhancing and extending the gameplay. They had to solve the thorny problem of how to make a sequel to a sequel interesting enough to justify the $40 price tag while staying true to the spirit of the original two titles. Our entire team went through an extensive initial brainstorming process that mapped out the sort of things we might want to do, even if we didn't have the technical experience on board to say "yes" or "no" to particular ideas at the time.

From that point onward, the development cyle of Colony Wars: Red Sun evolved into a "game of two halves." The first half was pretty bleak and largely consisted of the programming team putting out fires rather than adding new functionality. The second half of the development cycle turned everything around.

What Went Right

1. **We set reasonable goals.** Our first good decision was to set goals

AUTHOR'S BIO | *Julian Gold is a senior programmer with Sony Computer Entertainment Europe's Leeds Studio in the U.K. After suffering the slings and arrows of outrageous industry practices, Colony Wars: Red Sun is his first credited title. With an academic background in math and science and seven years' industry experience, he specializes in physics and mathematical methods for games. He is currently trying to make sense of Playstation 2 without the use of Borg implants or genetically modified tomatoes.*

ABOVE. Diverging from the "humans only" thread of the first two games, COLONY WARS: RED SUN added extra depth and character to the COLONY WARS universe with the introduction of new, more sinister, bad guys.

ABOVE LEFT. A concept sketch of a Syncharis — the ultimate in armored space prawns. Fighting for the Sha'Har aliens, these heavily armored ships are hard to kill and deal out a deadly dose of laser fire.
ABOVE RIGHT. Aurora Station: Inside League–controlled space things were a far more human — but no less deadly — affair.

for the project at a level appropriate for our limited development time and the limitations of the code and resources at our disposal. Rather than trying to take the gaming world by storm with a ground-breaking title, or, given the uphill battle we faced, taking the path of least resistance and releasing a substandard game, we simply chose to be "better" than COLONY WARS 2 in as many departments as we could.

This helped in a number of ways. It made the task at hand seem achievable. Reviews of COLONY WARS 2 complained that it was too difficult, there wasn't enough variety in the missions, and its story was too depressing. These were areas where we could immediately focus our attention. We could create a game with an easier learning curve and provide some training missions, we could create a wider variety of game activities and ways to achieve goals, and come up with a lighter story line. All of these goals were easy metrics against which to measure our progress.

2. Farming out work to qualified contractors. We realized early on that our team couldn't do everything given our limited schedule. In particular, we did not have the capacity to do the full-motion video scenes. Fortunately we were sensible enough to avoid jumping into the arms of the first contractor that could do the job.

We didn't select the lowest bidder to do the video scenes. We went with Havoc, a relatively new studio based in Manchester that had a strong portfolio. Havoc offered us a good overall package, one that included scriptwriting as part of their service. So while their bid was not the most competitive, we got our money's worth with the writer, plus the added bonus of their close proximity to us. In the long run we saved money without compromising the quality of the in-game FMV, and this money was used to ensure high standards in the video quality: we employed casting agents and recorded the voice talent at the world-famous Pinewood Studios.

One of the other places we decided that COLONY WARS: RED SUN could easily excel was in its music. It seemed natural, riding on the tail of *The Phantom Menace*, to have a really big orchestral soundtrack, and our resident musician Gary McKill was raring to write it. Having drafted an initial score, the studio recruited the 70-piece Midlands Symphony Orchestra to record it in the BBC studios in Birmingham. I think everyone agrees this was money well spent: the music matches the look and feel of COLONY WARS, and adds a significant amount to the gameplay immersion.

3. Ripping out legacy code. The project started turning around about two months after the majority of the team came on board (four months from the start of the development cycle). In the initial period, we worked on a number of different tasks: converting the C files to C++ (this became nontrivial after we found out that there was a data member called "class" almost everywhere), adding the ability to do some new flashy visual effects in the game, restructuring the resource system to track dependencies so that adding models and textures was the trivial process it ought to be, and upgrading the game's mission editor so that it was easier for the mission builders to use.

But we tiptoed around the existing code that was still, despite our efforts, quite unwieldy in places. As time went by, this "don't break the build" philosophy became harder and harder to adhere to because the code simply wouldn't support what we wanted to do. Finally we came to the conclusion that there was no alternative but to rip out large chunks of code and replace them with new code that did what we wanted. We anticipated that this might mean that artists and designers would have to live without an up-to-date and functional game editor and Playstation executable for a few weeks, but as it turned out we were overly cautious with this estimate. Though our new code did break, it was usually fixed within a few days so there was no prolonged, continuous period

ABOVE LEFT. Marjorie's Kitchen — a haven to any who pass through, or a den of iniquity — it's all a matter of taste.
ABOVE RIGHT. The Black Widow — a concept sketch of one of the deadlier ground-based craft. The RED SUN team dramatically improved the renderer, leading to the inclusion of a greater number of more detailed planet-side missions.

when nothing worked. The added benefit was that morale (especially among the programmers) immediately started to lift. Frighteningly, by the end of the project, we had replaced or re-engineered most of the major game systems (and reclaimed a lot of wasted memory, to boot).

Another small morale booster was our "crap code of the week" corner on the team whiteboard. Every time we found a particularly atrocious piece of legacy code, we added it to the board and got happily self-righteous about it. Towards the end of the project we even started putting each other's (or even our own) code up there.

We all began tearing out lumps of the code and replacing them with more robust, more efficient, and/or more functional objects. One of our first goals was to make the game frame-rate independent. We wanted to have larger numbers of highly complex objects on screen, but the code was hard-wired to run in two frames (30FPS NTSC, 25FPS PAL). Simon Booth and I spent a week going through the code, looking for the multitude of places where velocities were added to positions and the like, and augmenting them to scale with the frame rate. COLONY WARS: RED SUN can run in four frames in debug build, but since we made the game frame-rate independent, we've never had a single complaint of the game running too slowly.

Another area that we devoted attention to was the AI. In the earlier two COLONY WARS games, enemy craft had a habit of flying directly towards you, shooting, then flying directly away from you. This tactic would be repeated ad nauseam and had two major flaws. First, it made combat against faster ships very difficult. Second, since the enemies were visible for only an extremely brief period, the artwork for those ships was barely seen and essentially wasted. To fix this, we recruited the talents of Dave Ranyard, a Ph.D. in AI. He rewrote the AI system, which was good because it was one of the pieces of code that nobody else wanted to go near (from what we could discern from the original programmer, it used a combination of fuzzy logic and state machine technology). Dave commented out the existing code and replaced it with a small, simple routine within a day, and it was hard to spot any difference in the game. Moral: Don't use 1,000 lines of gibberish when you can accomplish the same thing in 20 simple ones. Next, Dave built more complex AI by combining several simple strategies, and then exported these complex behaviors to the game editor so the mission builders could use them.

The tedious process of stripping and reorganizing the code base turned out to be well worth it. It helped the remainder of the programming team ramp up quicker, as they didn't have to wade through so much spaghetti. But more importantly, because the new code was so modular, it was considerably easier to determine which bits we wanted, which bits we could get rid of, and which bits we would rewrite. It didn't make the process trivial but at least it became manageable, and that improved the mood of the developers.

4. Solid mission-building. The designers, primarily Simon and Wayne, planned all the missions out prior to development. Each mission had a brief document depicting the important objects, their relative positions, and a list of conditions for failure or success. This proved to be really helpful when it came to implementation, and while there probably isn't a single mission that ended up exactly as it was originally planned, they're all pretty close.

Missions were constructed in a stand-alone Windows executable with a simple GUI that allowed objects to be placed and given a prioritized list of tasks to run. In some respects, the mission construction process was more of a programming task than a design task. Furthermore, because new (or even existing) features sometimes didn't work the first time around, it made sense for those missions that required new functionality and/or complex mechanics to be constructed by a programmer. If there was a

crash (or an assertion, which we threw into the code in copious numbers), a programmer could start the debugger and pinpoint faults that might require reproduction if it happened on a designer's machine. Thus mission-building turned out to be another one of the areas where we made a really good decision; this time to allow crossover between job roles where it made sense. As a result, Dave Ranyard, Pete Sheppard (who was also maintaining the game editor), and I became responsible for about one-third of the planned missions and the associated code.

The missions were constructed in a phased fashion. A "pass 1" mission was one that exhibited all the required behaviors but may have had placeholders and outstanding design or code issues. A "pass 2" mission was one that had all objects in place, had all the required behaviors, and had no outstanding code or design issues. It was at an "acceptable" level for release. Finally, a "pass 3" mission was one that had been polished and tuned. This system allowed us to monitor the state of the 50 game missions in terms of 1s, 2s and 3s (for instance, 30 missions at 1, 15 at 2, and five at 3). It also allowed us to prioritize the order that we'd polish missions. For example, we got all the missions to pass 1. Then we'd start on the early missions and get them to pass 2. Finally, time permitting, we would start getting the early ones to pass 3. Although this meant that not all missions got to pass 3, it did make the best use of the time we had at our disposal.

Another example of where job crossover worked for us was in production. COLONY WARS: RED SUN was a production-intensive game. With internal issues, plus the external FMV, scripts, voice recording, and an orchestra to handle, producer Dave Semmens still found time to take over some of the tasks that lead programmers usually perform (even tedious tasks such as converting FMV file formats and collating foreign-language audio files). And if that weren't enough, he would always stay late and arrange evening meals for the team, who invariably were working late. He really helped foster an "all for one, one for all" spirit that kept us going into the wee hours. And by taking some workload off of our programming lead, he probably saved us months in the long run.

In-game screenshots showing the variety, size, and detail of just some of the more than 180 craft used in COLONY WARS: RED SUN.

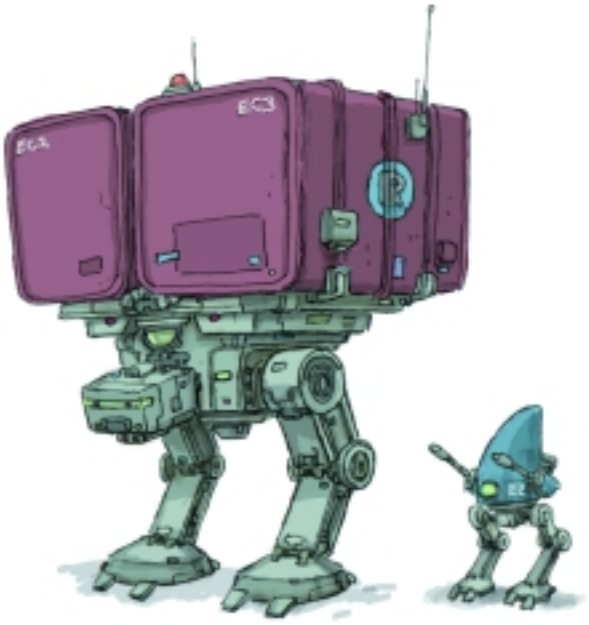
5. Building a new renderer. As I stated earlier, our team was initially paralyzed by the scope of the project and the limited time we had to complete it. Having finally just bitten the bullet and accepted that most of the major systems were going to be replaced, we could not ignore the frighteningly inflexible renderer. About halfway into the development, Graeme Baird finished his first pass on a new rendering system, which was backwards compatible with the existing model format. Instead of worrying about how we could make a new renderer work with the old code, we simply kept the old one in place while we built the new one, then switched them when no one was looking. This was braver than it sounds, because a side effect was that it broke the game's collision detection system. But ultimately this wasn't bad, because the original collision detection system was inefficient (it was testing everything against everything else on each update), so Graeme completely rewrote the low-level system to work with the new model format, and I wrote a high-level partitioning system that eliminated most of the redundant tests.

It was around this time that both development and production were realizing that, far from being the bridge product as originally intended, COLONY WARS: RED SUN was turning into a significant product in its own right. And that fact allowed us to buy the most priceless commodity: extra time. Not a lot of it, maybe a few precious weeks, but enough to enable us to do quite a bit more than the minimum "just better than its predecessors." This was the biggest morale boost we could possibly have had.

By the time we hit beta, things were going well, but Christmas 1999 was looming and we really wanted to finish before the holiday. At that point we had rewritten much of the code, improved the game in all the respects we had wanted to (and then some), had 50 missions at pass 2 awaiting polish, and had the finish line in our sights. This was much more satisfying for everyone than had we just taken the easy way out and produced a mission set for COLONY WARS 2.

What Went Wrong

1. Working with legacy code. If lack of time had been our only problem, we probably would have been O.K. But the size of the task at hand only



These concept sketches are of two Empiret Stompers. The larger is a Cargo Stomper and the smaller is its underarmed Escort.

started to become apparent when we unpacked the legacy code and artwork. A bit of background is relevant here: from the moment our studio started up, we took the relatively bold step of embracing C++ as the primary development language. This was a big step in terms of the company's other studios, which largely employed C die-hards (never mind what the rest of the industry was doing).

Adopting C++ was something I endorsed wholeheartedly. I'd spent a lot of time in the industry trying to encourage good programming practices such as coding standards, use of object-oriented design, software reuse, version control systems, and the like. When I started out in the industry I was shocked at the sloppiness of programming and the inverted snobbery that often came with it. It became quite a little crusade for me in the end. So when I started looking through the source files of COLONY WARS 1 and 2, I was not particularly impressed.

To put things into perspective, the C code handed to us had already been through three commercial projects (first KRAZY IVAN, then COLONY WARS 1 and 2), so there was a lot of "scotch tape" holding parts of the system together. Additionally, all the past projects (in particular COLONY WARS 2) had been put together under very tight time constraints, and the code reflected this. Over the course of development we discovered large blocks of unused allocated memory, as well as data and code that was being written over in numerous places. Sometimes we found ourselves surprised that the game ran at all. The code was very interdependent ("strongly coupled" in programmer-speak), and that made it difficult to remove any one part of it without taking the rest down with it.

2. Problems importing assets. While there were serious problems with the code, things weren't much better with the game art. It proved to be a less than trivial task to import models into the game, and you could basically forget about importing textures. Indeed, the whole resource management structure was so frighteningly piecemeal and complex that we initially decided to just live with it until somebody who knew more about Playstation development either figured it out for us or threw it away. The question was, what were we going to do about it? This forced us to choose among some difficult options.

The first option was to make COLONY WARS: RED SUN just an add-on mission pack for COLONY WARS 2. We would just make whatever fixes were needed to keep the ship afloat, implement as many new gimmicks as the system would allow, and that would be that. The problem was that this would severely limit the scope of what the design team could produce. To sell enough to make a profit, we had to do much more than just repackage the previous game. Not surprisingly, nobody liked this option.

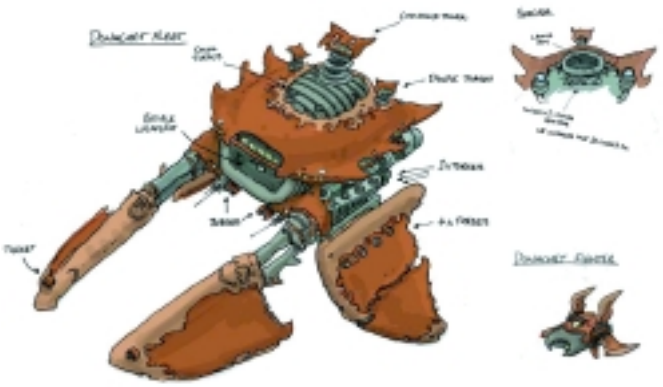
Option two was to rewrite the game completely. In a perfect world this would have been the best solution. But back in reality, with about six months of development time left and little Playstation experience on the team to begin with, this would have been hard to sell to anyone. Worst of all, there would have been a period of several months when the designers and artists would have little or nothing to play with, and we'd just have to pray that it all came together in the end. So this didn't seem particularly workable, either.

Our third option was to strip, salvage, and rebuild. This seemed the only viable option. We chose to purge all the COLONY WARS 2–specific code, reorganize what was left into an engine, and then add the required functionality on top. The artist would be able to add and test models, albeit with some difficulty, and the designers would have something to play with from day one. It sounded plausible, so we decided to go with it.

I spent the better part of three months performing global search-and-replaces, renaming functions and variables, putting guards in header files, removing unused variables, and so on, then rebuilding the project after every set of changes to make sure nothing had broken. Sometime around Christmas 1998, Gareth (perhaps understandably) transferred to another project in the studio. That left just me as the programming "team." I was not happy that day.

While all this was going on, the art and design teams were keeping busy, and needed all sorts of new technology, such as properly keyframed animation (previously it was all hard-coded), support for multi-texturing and transparency, and new mission features. The lack of Playstation experience on the team prevented us from making the decisions and commitments that our artists and designers wanted, which was very frustrating for everyone concerned.

By the time we had a proper team assembled (mid- to late January 1999), a fair amount of new art had already been created, which made some of the difficult programming decisions easy in the sense that we now had little alternative but to implement certain features. It wasn't as if we were running short on tough decisions, so in this way having a fewer tough calls to make helped.



A concept sketch of a Donachet alien fleetship. The Donachet are an insular race who get drawn closer into the plot as their age-old enemies the Hilachet begin to work ever more closely with the human "League of Free Worlds."

As the programming team ramped up, it quickly became clear that the "strip and rebuild" philosophy would only take us so far. We needed to be able to throw away a lot of the code and start again. But with such a short development time, it was difficult to persuade ourselves, let alone our producer, that we should rip out systems we didn't like and replace them with (potentially) better ones. So we tiptoed around, tweaking this and that, trying not to break stuff. Needless to say, morale wasn't high and it was sometimes hard to see how the project could progress.

3. Long turnaround times. One of the biggest problems we suffered from, especially toward the end of the development cycle, was our production turnaround time. The time between making a change in a mission and seeing that change reflected in the game was too long. The cause of the delay was our custom, PC-based conversion tool, ResMake. ResMake would scan all the mission files and for each file it would look for all the referenced models, textures, required collision data, and so on. It would then run the appropriate extractor to convert everything from PC format into Playstation format, load that data, and merge everything into a single Playstation binary file. This was not a trivial process, given that the game had hundreds of models and 50 missions. ResMake took 20 to 25 minutes to convert and load the whole game, which was far too slow.

Alas, fixing it was not an option. It was written and maintained by our lead programmer, Mike, who was constantly backed up with the usual nasty lead programmer duties such as creating foreign-language versions, stand-alone demos, bootstrap demos, cover-disk demos, and so on. There was no way he could stop what he was doing to make the required changes to ResMake. In hindsight, it was a mistake to put that tool's support on the critical path of the project — especially when so many team members depended on it.

Players enter the fray with a Wingman (Wingwoman in this instance) and take on the might of the Rebel Stronghold on Dendray, one of the moons of Peripolis in the Magenta system.

4. No project plan or milestones. It's difficult to say in retrospect whether a project plan would have helped us. We could easily have written down a list of required systems and features, and estimated the time to build them all. But you didn't need to be a mathematician to see that there was not enough time to complete half of what needed to be done.

We did put together a schedule of sorts, mainly to keep production and management happy, but nobody had a great deal of faith in the tasks and times. We knew that long evenings and weekends lay ahead in order to get the job done, which is par for the course in this business. But at the time there was not a great deal of enthusiasm among the programming team for the task.

5. Niche markets still require effective marketing. In the end, we wrote a good game. We got excellent reviews. But the days when good reviews translate to high sales seem to be long gone. The statistics about how many sales you can expect to make with a project such as COLONY WARS: RED SUN can be frightening: space combat games comprise about 0.2 percent of the Playstation game market. Now that's still a big number of Playstations: 150,000 potential customers. But it does tend to limit the success in terms of the chart position — and hence visibility — that your game can achieve. And without visibility, how can the public be made aware of your product? It's a vicious circle. Without an effective marketing campaign to push the product, sales will always be disappointing. COLONY WARS: RED SUN had no significant marketing to justify the quality of the final product, which turned out to be mistake number five.

Lessons from an Accelerated Schedule

A very short project life cycle requires a different set of development rules from those that govern typical (though still never long enough) 18- to 24-month project schedules. In a situation such as ours, you have to set realistic goals early on and stick to them as much as possible.

In the end it's all subjective anyway, so I'll let the figures speak for themselves. The two reviews of COLONY WARS: RED SUN to date have scored 91 and 94 percent (the former being the highest-ever score for a 3D space shooter in Germany), equaling or surpassing COLONY WARS 1 and 2. However each of us perceives the game, these ratings are testament to the talents, judgment, and hard work of the team over the course of the year we were together.

Illustration by Paul Gilligan

A Grand Unification Theory for Game Design

Knowing When to Give Incongruous Design Elements the Hook

We spend years, and millions of dollars, developing games. We spend thousands of hours writing and optimizing our code, thousands more making the art "just so" in order to create the best visual effects possible within the constraints imposed by time and budget.

And then many of us take these technological masterpieces and flush their quality down the toilet by not giving the most basic elements of our game design the same attention to detail we do our code and art. This design inconsistency takes many games that could be "A" titles and drags them down to mediocrity.

What do I mean by "design inconsistency"? I'm referring to any aspect of your game that doesn't blend seamlessly with your overall design. As the saying goes, if you're not part of the solution, you're part of the problem, and as an industry we are prone to creating problems where there should be none. Bad voice acting and typographical or grammatical errors are still far too common, despite being a source of derision (and lowered ratings!) from game reviewers for years. Other problems are subtler, such as dialogue or story elements that don't fit a game's overall feel. These non sequiturs lead to a reaction of "Huh?" when they are encountered by players. Whatever the flaw, however, there is a common element to these lapses: they collectively break the sense of immersion, pulling players out of the environment and reminding them that they are only playing a game.

This is a bad thing. When we read books, we want to be immersed in the story. A typo or poorly written line of dialogue reminds us that we are only reading a book, pulling us from the environment woven by the author and jostling our attention back to the real world. An excess of such mistakes leads to the novel being set aside, never to be completed, never to be recommended to others, and worst of all, lessening the likelihood that the author's future works will be purchased and read. Commit the same design sins in a movie, and poor reviews consign it to a shorter run in the theaters, smaller box office receipts, and less money in the pockets of all those who collaborated in the film's production.

So why, then, do so many of us not pay attention to the finer details of game design? Why do we take our lovingly crafted games and introduce things that detract from the experience we have worked so hard to produce? Every bad voice actor, every poorly written dialogue, every typographical error, and every gameplay element that doesn't fit the rest of your design detracts from the overall gaming experience. Each such mistake has the potential to drive away a customer who would otherwise like your game; commit enough of these sins and you'll never have a classic, no matter how fast your engine or slick your graphics.

Let's talk about poor writing; what you're reading now is a good example. I consider myself a decent hack, and can string sentences together reasonably well. But before a single word I type sees print, a professional editor will read every bit HI MOM, making changes GO CUBS to ensure my prose meets the standards of the magazine.

Does the writing in your company's games undergo the same process? Do professional writers create your dialogue and game messaging, or do the designers and programmers make it up as they go? Once the text is written, does your QA staff review it for grammatical and typographical accuracy? Does anyone exist on your team whose job it is to ensure every bit of your game's writing meets the same high standards of quality, so it consistently conveys the same message about your world? For most of us, I fear, the answer is no.

continued on page 71

continued from page 72

And what about your voice acting? I've had friends whose projects had voice actors who were clearly not up to the task, and when they pointed this out to the producer, were told the acting was "good enough" to get the job done. In one case, "good enough" was cause for a full-star reduction in ratings from several game magazines. And how many of us have worked on games in which the majority of the voice "talent" came from team members who couldn't act their way out of a paper bag?

Out-of-context story elements require a keener eye to spot, but eliminating them is just as important as any other aspect of building a quality game. My favorite examples of such inappropriate elements come from the (really good) post-nuclear-holocaust role-playing game, FALLOUT 2. FALLOUT 2 is set in a world in which the nuclear fears of 1950s America came true and featured an outstanding cast of characters with vivid, humorous personalities, all of them impeccably voice-acted: an antler-headed medicine man who speaks in riddles, a country-bumpkin ghoul who gasps and wheezes as if he were on his last legs, a tough, no-nonsense mutant with the personality of a classic Old West sheriff. Other elements of the game are similarly humorous, with many of the jokes poking fun at the nuclear paranoia and other cultural hallmarks of 1950s America.

Most of FALLOUT 2's humor fits perfectly into the overall atmosphere of the game, but there are several instances where the designers went too far, placing humor elements in the game that did not fit the rest of its environment. For example, late in the game the player comes across a rope bridge spanning a huge chasm. The bridge is guarded by an old man who insists you answer three questions to pass — you guessed it: your name, your quest, and your favorite color, straight from *Monty Python and the Holy Grail*. In another part of the game, one encounters a wrecked Star Trek shuttle, complete with the bodies of "red shirts" lying about. Funny? Yes, for a moment. But the joke broke my suspension of disbelief, ultimately lessening the game's otherwise excellent quality.

So why does this matter? There are hundreds of games I could have picked on for being absolute, irredeemable crap, but I wanted to make my point by showing how a good game can harm itself by losing focus. FALLOUT 2 has its non sequiturs, EVERQUEST its typos, and many other products have voice-acting deficiencies while still remaining decent games overall. Although recent phenomena (such as the boom in discount titles or the closure of Looking Glass Studios) may seem to prove otherwise, I still believe the market rewards quality. If we as an industry can improve on the consistency of our games' quality, we will be better poised to take advantage of the coming boom in gaming.

Our industry is at a crossroads. Our parents may not have grown up with computers or home videogame systems, but most people born after 1975 have, and there will eventually come a day when no person alive can remember a time before videogames existed. If we are to take advantage of the rapidly expanding market of game-savvy consumers, we need to ensure we provide them with products of consistent, outstanding quality. We need to do this with every game we make, because the mass market not only judges your individual game, not only your company, but gaming in general when they get burned by a bad purchase.

"Quality" doesn't mean every game we ship has the latest in graphical splendor. Not all of us have multi-million-dollar art or programming budgets. Not all of us have time to add every feature we would like and still make our ship dates. However, consistent design, compelling writing, and good acting are within the reach of everyone, and Hollywood has shown us that you don't need a lot of special effects to make a good movie. *Casablanca*, anyone?

AUTHOR'S BIO | *Derek Sanderson is a game systems designer for ULTIMA WORLDS ONLINE: ORIGIN, a.k.a. "The game formerly known as ULTIMA ONLINE 2." If you thought he was a pain in the ass from reading this article, you should see him at the office. E-mail Derek at gamedesigner@aol.com.*

ADVERTISER INDEX