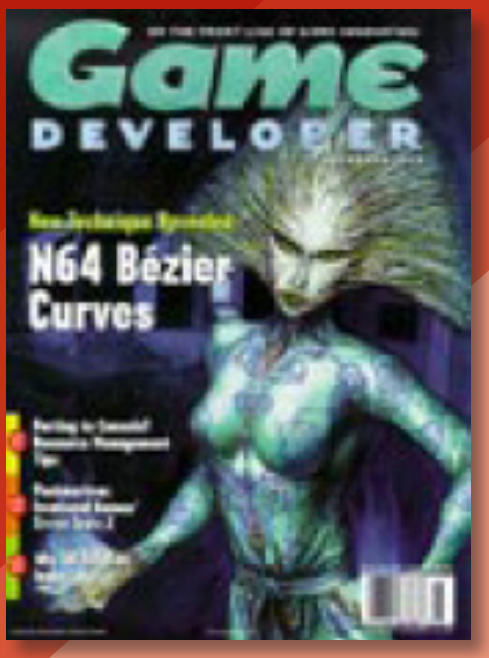
gd

GAME DEVELOPER MAGAZINE

NOVEMBER 1999



UBM
Tech

Can Subsidized Hardware Save PC Gaming?

With one of the largest console launches ever just behind us and at least two more looming on the horizon over the next 18 months, many people are speculating about the features of these next systems. Upon seeing the impressive feature sets of the new consoles, some have raised questions as to the long-term viability of the PC as a gaming platform. It's tough to look at a console spec sheet, read that it has broadband Internet access, amazing graphics and audio capabilities, a keyboard, DVD support, support for electronic software distribution and backwards compatibility, and not see that the PC faces stiff competition. The traditional strengths of the PC are being co-opted by consoles. I for one don't believe PC gaming will go away, but the platform must confront these challenges head-on.

To grow the PC market, prices must continue to drop. Fortunately they are, and it's one reason I'm bullish on the PC's future. One big reason for recent drops in PC prices is the direct result of marketing campaigns by ISPs like Compuserve and AOL. Compuserve, for example, is subsidizing the cost of Compaq, HP and eMachine entry-level systems to the tune of a $400 rebate at purchase. In return, these consumers (many of them first-time PC purchasers) agree to subscribe to Compuserve for three years at $21.95 per month. These ISPs are gambling that it's better to underwrite the cost of these new machines today and recoup that investment through extended online service agreements. The ISP gets reimbursed for its up-front investment from these multi-year service agreements, it gets "content" in the form of new chat participants, more eyeballs for which it can sell advertising space on its service, and other assorted benefits. If it persuades these indentured servants — I mean, customers — to stay with Compuserve after the agreement is terminated, so much the better for the ISP. The bottom line is that as I write this, you can buy a brand new 400MHz eMachine with 15-inch monitor and a color printer for about $90 more than a Dreamcast.

While I'm glad that more PCs are working their way into homes — it certainly will grow the base of casual gamers — there are a couple of aspects to these incentives that don't bode well for the PC game industry. First, these rebate programs are seeding households with machines barely equipped to play today's games. The latest graphics and audio hardware won't be found in a $289 computer, and without these capabilities, the systems offer little in terms of a cutting-edge gaming experience. Second, long-term service agreements with online services like Compuserve could hamper the growth of broadband access to the Internet, and enabling broadband access is essential for growing the online gaming market.

It would be interesting to see a major game publisher like Electronic Arts step up and subsidize a line of high-end game PCs, targeting veteran PC owners and hard-core gamers. In return for the rebate at purchase, these consumers might agree to buy a certain number of games from EA over a span of years (the Columbia House music club model), or sign a multi-year subscription to a persistent game world like ULTIMA ONLINE. The latter option is especially intriguing, since the risk to publishers of these games is high — developing and maintaining a persistent world is expensive, they are more difficult to manage, and more rides on their success than with traditional games. A publisher could ensure that when the game was built, some percentage of players would be locked into the game. A guaranteed revenue stream over a period of years looks good on the books, and in the hit-or-miss world of game publishing, that's important to Wall Street.

Of course, there's nothing stopping any of the console manufacturers from launching a similar rebate program to entice their customers. It may simply be a matter of who strikes first. ■

ON THE FRONT LINE OF GAME INNOVATION
Game DEVELOPER

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

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

EDITORIAL

Editorial Director
Alex Dunne *adunne@sirius.com*

Managing Editor
Kimberley Van Hooser *kvanhoos@sirius.com*

Departments Editor
Jennifer Olsen *jolsen@sirius.com*

Art Director
Laura Pool *lpool@mfi.com*

Editor-At-Large
Chris Hecker *checker@d6.com*

Contributing Editors
Jeff Lander *jeffl@darwin3d.com*
Paul Steed *psteed@idsoftware.com*
Omid Rahmat *omid@compuserve.com*

Advisory Board
Hal Barwood LucasArts
Noah Falstein The Inspiracy
Brian Hook Verant Interactive
Susan Lee-Merrow Lucas Learning
Mark Miller Harmonix
Dan Teven Teven Consulting
Rob Wyatt DreamWorks Interactive

ADVERTISING SALES

Western Regional Sales Manager
Jennifer Orvik e: *jorvik@mfi.com* t: 415.905.2156

Eastern Regional Sales Manager/Recruitment
Ayrien Houchin e: *ahouchin@mfi.com* t: 415.905.2788

International Sales Representative
Breakout Marketing e: *breakout_mktg@compuserve.com*
t: +49 431 801703 f:+49 431 801797

ADVERTISING PRODUCTION

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

MILLER FREEMAN GAME GROUP MARKETING

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

CIRCULATION

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

INTERNATIONAL LICENSING INFORMATION

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

Miller Freeman
A United News & Media publication

CEO/Miller Freeman Global Tony Tillin
Chairman/Miller Freeman Inc. Marshall W. Freeman
President & CEO Donald A. Pazour
CFO/COO Ed Pinedo
Executive Vice Presidents Darrell Denny, Galen A. Poss, Regina Starr Ridley
Sr. Vice Presidents Annie Feldman, Howard I. Hauben, Wini D. Ragus, John Pearson, Andrew A. Mickus
Sr. Vice President/Development Solutions Group KoAnn Vikören
Group President/Division SF1 Regina Ridley

BPA INTERNATIONAL

www.gdmag.com

BIT Blasts

News from the World of Game Development

New Products: Metacreations introduces Carrara, NxN rolls out Alienbrain, and Criterion unveils its next generation of Renderware. **p. 7**

Industry Watch: Dell snubs ATI, Microsoft shows off its new console, Sony divulges more PSX2 secrets, and Aureal busts out with boards. **p. 8**

Product Review: Jeff Lander sings the praises of multi-resolution meshes as he reports on Digimation's MultiRes Toolkit and Plug-in for 3D Studio Max. **p. 10**

New Products

by Jennifer Olsen

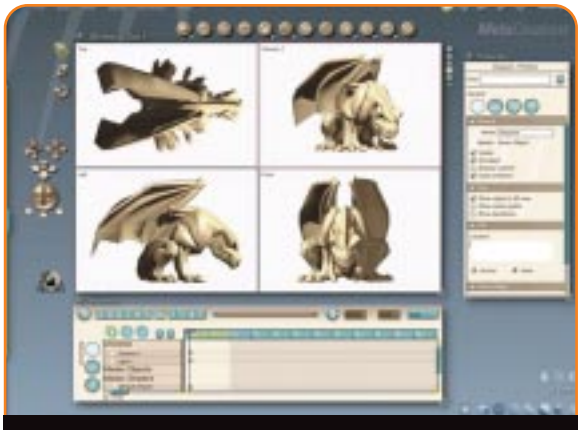
A New Chip off the Old Block

METACREATIONS has unveiled Carrara, its newest entry into the fray of "bargain" 3D modeling and animation packages. The name is appropriate enough: Italy's Carrara marble has been prized for its unsurpassed quality since ancient times, back when a huge slab of marble was the original modeling environment for 3D artists.

Carrara is actually the marriage of Metacreations' Ray Dream Studio and Infini-D products. So what separates it from the rest of increasingly crowded pack of 3D modeling and animation programs? It has much the same laundry list of features and effects as many of its competitors, including multiple renderers, Direct3D and OpenGL support, importing and exporting of most industry-standard 2D and 3D file formats, scads of shaders and presets, and an SDK for customization. However, one feature that sets it apart is its clean, attractive user interface, which lacks the screen-clogging blizzard of buttons characteristic of certain wildly popular high-end packages.

Carrara will sell for $499 and run on Windows 95/98/2000/NT 4.0 and Macintosh platforms.

- **Metacreations Corp.**
  Carpinteria, Calif.
  (805) 566-6200
  http://www.metacreations.com



Carrara's modeling interface: its lack of clutter may be inviting to some, limiting to others.

X-File Management and More

NxN SOFTWARE has announced Alienbrain, the successor to its groundbreaking asset management tool tailored specifically for game developers, MediaStation. Remember back in high school when you wrote your first game? Maybe you had a few dozen files and kept track of them on the back of your physics homework. Whatever system you had, it wouldn't work on today's development projects which now comprise thousands of files and mountains of media to manage.

The folks at NxN feel your pain, and Alienbrain is their antidote to the mind-boggling complexity of managing a game development project. The client/server system includes four modules to track file management, version control, process automation and project tracking across different user groups: "Genius" is designed for the artists on your team, "Intelligence" for the programmers, "Control" is the command center for project administrators and tool developers, and "Knowledge" is for producer-types. While the nomenclature won't settle any debates about who the real brains on your team are, each module contains features and functionality unique to the needs of its users.

Alienbrain's server systems require Windows NT 4.0 or later, and clients require Windows 95/98/2000/NT 4.0. Pricing was not yet finalized at press time, but the servers are expected to cost in the neighborhood of $4,900, with client systems priced at around $1,900. NxN also offers flexible volume pricing packages.

- **NxN Software AG**
  Munich, Germany
  +49 (89) 27-32-24-0
  http://www.alienbrain.com

Gearing Up for the Next Generation

CRITERION SOFTWARE has revealed the third generation of its multi-platform 3D development toolkit, Renderware. By now we've all gotten an idea of what the near future of 3D gaming holds both for PCs and consoles. As demands on developers increase, platforms diversify and pressure builds to decrease development lead time, more developers may be considering outside resources for help.

Renderware is based on a streamlined plug-in architecture that allows developers to mix and match functionality by overloading the pipeline with their own tools (physics or collision detection, for example,) when they so desire. The PC version supports Glide, OpenGL and Direct3D, with a device-independent architecture that will enable easier porting to consoles and tomorrow's digital TV platforms.

Renderware 3 will be available by year's end for PC, Playstation 2 and iMac at $1,000 per programmer per platform per year with no royalties, and third-party plug-in development is in full swing. Linux, Dreamcast and Nintendo Dolphin versions of Renderware are also being considered.

- **Criterion Software Ltd.**
  Guildford, Surrey, U.K.
  +44 (0) 1483-406200
  http://www.renderware.com

Industry Watch

by Alex Dunne

DREAMS DO COME TRUE. Sega's Dreamcast launch proved to be better than the company hoped, with preliminary figures indicating that $97 million in sales were rung up around the U.S. on 9/9/99. Sega claims it was the biggest 24 hours in entertainment retail sales, easily surpassing the $28 million that *The Phantom Menace* took in on opening day last May.

PSX2 NEWS. Carefully timed to coincide with the Dreamcast launch, Sony stole some of Sega's thunder with a number of Playstation 2 announcements. The company revealed that the PSX2 will debut in Japan next March (three months later than originally planned, possibly due to chip manufacturing problems), and in the U.S. and Europe next fall. Japanese consumers will have to cough up 39,800 yen ($365 at press time) for the system. Sony also revealed that it will distribute games via the Internet, made possible by the PSX2's broadband support and an upcoming hard drive Sony will sell. To support this new means of distributing console titles, Sony is creating an electronic transaction system, and an e-distribution server. In developer news, Sony will give PSX2 developers tools that support the regular Playstation programming/debugging mode as well as a new workstation mode for creating PSX2 graphics, all on the same system.

ENTER MICROSOFT... At ECTS, Microsoft demo'd its upcoming console (code-named the "X-Box") to various developers and analysts. As we go to press, no release date for this console has been hinted at, and product specs are sketchy. But the word on the street is that the X-Box will be based around a 500MHz Intel chip, the Nvidia GeForce 256, a DVD drive, a multi-gigabyte hard disk, and of course, some flavor of Windows. Who will produce this console? Not Microsoft, apparently — Dell, Gateway and Samsung have been lined up as manufacturers.

Sony took advantage of the Dreamcast launch to divulge more tidbits about the Playstation 2.

IT'S TEN NO MORE. Total Entertainment Network (TEN) ditched its name and its target market, deciding that the casual game market is more lucrative than its previous focus on hard-core players. The company, now called Pogo.com, is focused strictly on card, trivia, board and other "family" games. The site has more than 3.5 million members, and has lined up has distribution relationships with @Home, Alta Vista, Cnet, Excite, Go, Netscape, and Sony.

BE LINES UP TITLES. Be Inc. and Monolith announced that SHOGO: MOBILE ARMOR DIVISION will be brought to the BeOS. SHOGO will be developed and published for BeOS by Wildcard Design. At ECTS, Be showed CIVILIZATION: CALL TO POWER, CORUM 3, and QUAKE 2 running on its operating system.

DELL DECISION HURTS ATI. ATI acknowledged that Dell's recent decision go with Nvidia chips in its OptiPlex computer line will cost ATI $10 million in sales per quarter. The company still expects to meet fourth-quarter sales projections when it reports results on October 21, but that didn't quell some panicked investors, and trading of ATI's stock was halted on both the Toronto and Nasdaq exchanges after the company revealed the cost of that lost deal. ATI points out that it will still supply Dell's notebook PCs, and that its relationship with the big computer company is still strong.

AUREAL LAUNCHES CARDS. Aureal entered the board business and is shipping two new Aureal-branded sound cards, the Vortex SQ1500 and the Vortex2 SQ2500. The new cards are marketed under the Aureal name by I/OMagic Corporation, and are supported by a multimillion dollar marketing campaign. Based on Aureal's AU8810 processor, the SQ1500 supports A3D 1.0 and features a 512-voice wavetable synthesizer. The SQ2500 is based on a new version of the Vortex2 AU8830 processor and supports A3D 2.0 with a 576-voice wavetable synthesizer. ■

UPCOMING EVENTS CALENDAR

1999 GDC RoadTrips

OGDEN ECCLES CONFERENCE CENTER
Salt Lake City, Utah
November 1, 1999

THOMPSON CONFERENCE CENTER AT THE UNIVERSITY OF TEXAS
Austin, Tex.
November 3, 1999

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

Software Development East

WASHINGTON CONVENTION CENTER
Washington, D.C.
November 8–12, 1999
Cost: variable
http://www.sdexpo.com

RE:Play Real World Conference

TISCHMAN AUDITORIUM AT THE PARSONS SCHOOL OF DESIGN
New York, N.Y.
November 13, 1999
Cost: free
http://www.eyebeam.org/replay

Comdex Fall

SANDS EXPO & CONVENTION CENTER
Las Vegas, Nev.
November 15–19, 1999
Cost: variable
http://www.comdex.com

Digimation's MultiRes

by Jeff Lander

Scalable 3D graphics has been a major event in 3D graphics this past year. With so many different gaming platforms and such a variety of graphics cards, making content that performs similarly on all systems is a real challenge. Game developers have commonly used different level-of-detail models to balance the performance. However, creating LOD models is a time consuming and tedious project for any game artist. Furthermore, dynamically changing between LOD models during the game can lead to annoying popping as they switch. Graphics researchers have been promoting the idea of continuous LOD algorithms for 3D models. In these algorithms, the model will smoothly scale from very high to very low polygon counts. In fact, Stan Melax wrote an article in this magazine about implementing a continuous LOD system ("A Simple, Fast, and Effective Polygon Reduction Algorithm," November 1998). However, creating a system that handles all the lighting and texture information and smoothly integrates into your production is a task that could easily tie up development resources for some time.

I DON'T HAVE THAT KIND OF TIME. With this in mind, I looked with interest at several commercially available continuous LOD systems at Siggraph 1998. One of these was a very interesting system developed by Intel. At the time, the project was not quite ready to be a product, but looked very promising. Well, at the GDC this year, Intel unveiled the fruits of this labor. They have teamed up with Digimation to produce the MultiRes Software Toolkit and Plug-in for 3D Studio Max.

The MultiRes Plug-in enhances 3D Studio Max by providing a method for reducing a high polygon mesh to a lower polygon count mesh. In this way, the plug-in is similar to the Optimize routine that is built into 3D Studio Max. However, MultiRes is quite a bit more powerful. For example, you can set exact polygon count targets as well as a reduction percentage. Also, MultiRes does an excellent job of preserving texture coordinates and vertex normals. Artists are even able to fine-tune the reduction algorithm by a variety of controls as well as selecting the vertices not to remove.

As a modeling tool alone, this provides a pretty powerful and useful way for artists to craft content. However, the real power of MultiRes is realized by creating a multi-resolution mesh. This special mesh file contains all the information needed to create a mesh that can dynamically scale from its full resolution down to 1 polygon composed of 3 vertices. This MultiRes mesh can then be used directly in your game as scalable content. It can also be exported into the MetaStream 3D format that is used with tools from MetaCreations to view scalable 3D models on the web.

FIGURE 1. *The bottom dinosaur's face count has been reduced dramatically, but looks fine from a distance.*

You can see an example of MRM in action in Figure 1. The first image is the original mesh at 6,126 vertices or 11,766 faces. I then reduced this down to just 64 vertices for 120 faces. Obviously, this looks pretty bad up close, but when the object gets far away it looks fine. The MRM algorithm preserved the outline of the arms and legs. This is where real-time use of MultiRes really makes a difference. A herd of these animals at full resolution would grind any system to a halt. But a herd of them at 120 polygons is perfectly reasonable.

HOW DO I USE IT? If you are a 3D Studio Max user, the plug-in couldn't be any easier. You simply select your model and select the plug-in from the Modifier panel. You can then "Generate" the MultiRes mesh and start interactively reducing the vertex count in the model. The default generation options do a good job of mesh reduction for many models. However, you can really fine-tune the operation.

The "Vertex Merging" option allows you to determine whether or not unconnected areas of the mesh will be merged as the reduction proceeds. You specify the maximum distance in 3DS Max units that the algorithm will consider for merging. This can be useful if your mesh is composed of parts that are not topologically connected.

"Boundary Metric" gives you the options with respect to any material changes in to model. It will then try to avoid collapsing vertices that cross these material boundaries.

Most of the time, the algorithm along with these options will allow you to create a good mesh. However, there are times that you will want to select vertices that you do not want to collapse in the model. Perhaps there is a feature in the model that is distinct and you wish to be preserved. You can select the "Maintain Base Vertices" option and then select the vertices you wish to preserve. These vertices will then be maintained throughout the reduction process.

Jeff Lander is always trying to find a tool to make his life easier and cut down on unnecessary work at Darwin 3D. If you can recommend any nifty tools, pass them on to jeffl@darwin3d.com.

The final option determines how the normals in the model will be handled. While vertices are removed, the topology can change pretty dramatically. You can simply use the original vertex normals throughout the reduction. Otherwise, by setting the "Multiple Normals per vertex" option, the system will create new normals based on the surrounding faces as the model reduces. This option comes at the cost of increasing the number of update records that must be recorded. Whether or not this is needed depends greatly on your application and models.

That's all there is to it. Once you are happy with the reduction, you can save it out, ready to use in your application.

BUT I DON'T USE MAX. If you don't use 3D Studio Max, you won't get the benefit of this nifty plug-in. However, the benefits of MultiRes are still available to you. The MultiRes Software toolkit contains all the functions you will need to create a scalable mesh. You simply set the parameters for the reduction and submit a structure containing all the vertices, normals, and faces in the model to the `GenerateMRM` function.

I found it very easy to take models created in Softimage and convert them into a MRM by modifying the Digimation sample viewer. This would be easy to do for any polygonal model.

SO HOW DO I USE IT IN MY GAME? Now I have a nice MRM all ready to go and I want to use it in my game application. The examples provided with the Software Toolkit make this easy. There is both a Direct3D and OpenGL example of working with a MultiRes mesh.

One thing developers will appreciate is that while the MRM generation functions are in a Dynamic Link Library (DLL), all the run-time code needed to display and manipulate the meshes are straight C. You need no extra libraries to ship with your project. This also makes it possible to use the MRM technology on game consoles.

Also, as the algorithm works by changing the connectivity of the meshes, the actual vertices are left alone. This means that the MultiRes algorithm can work with most animation schemes such as skeletal deformation, morphing, or even QUAKE-style mesh flipping.

OTHER FEATURES. The MultiRes Toolkit also offers another very valuable feature. In order to render polygonal meshes in the fastest way possible, many 3D graphics cards prefer to receive the meshes as triangle strips. These strips can be tricky to create and often require custom tools to generate them.

MultiRes provides a way to generate triangle strips from a polygonal model. However, since the model's topology changes, as the level of detail changes an initial triangle strip would become invalid. To address this issue, the MultiRes Toolkit provides a way to generate triangle strips on the fly. Very cool...

WHAT'S THE BOTTOM LINE? The 3DS Max MultiRes plug-in is $295. By itself, this plug-in may be of use to Max modelers who wish to have a better polygon reduction tool or want to generate MetaStream objects for web viewing. If you don't use Max or really want the game interactivity, this is probably not for you.

The MultiRes Software Toolkit includes three license copies of the Max plug-in as well as all the libraries and code needed to generate and display continuous LOD meshes. The cost of the toolkit is a flat fee of $5,995 per finished game title. When you consider all the technology included and the amount of development time it would take to create this functionality, this seems like a great deal to me.

Obviously, I'm not the only one who thinks this is interesting. Both Valve with TEAM FORTRESS 2 and Pandemic Studios with DARK REIGN 2 have licensed the MultiRes Toolkit for their upcoming 3D titles. I expect many more to follow.

MultiRes is the first commercial project to come out of the collaboration between Intel and Digimation. I certainly look forward to other products that come of this partnership. ■

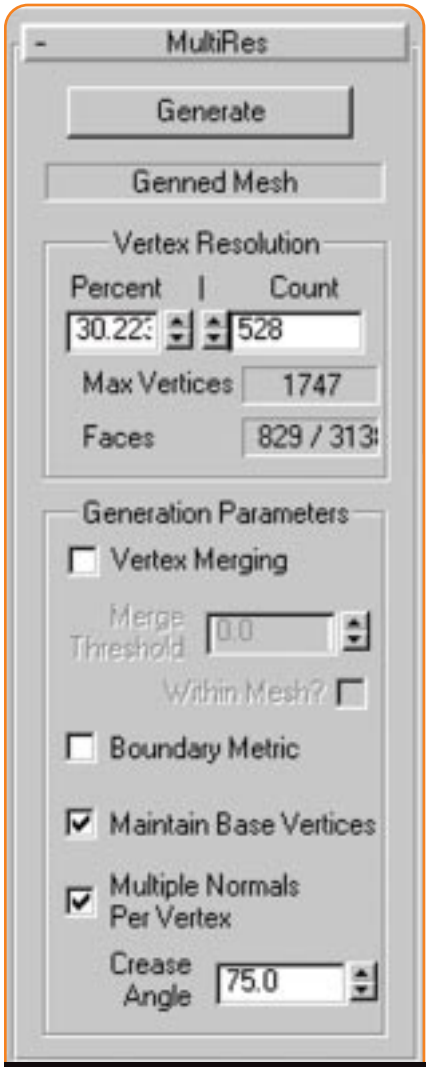


FIGURE 2. *The MultiRes user interface.*

MultiRes: ★★★★★

Digimation Inc.
St. Rose, La.
(800) 854-4496
http://www.digimation.com

Price: Plug-in is $295. Software toolkit is $5,995 per finished game, including three copies of the plug-in.

Software Requirements: Windows 95/98/2000/NT 4.0; 3D Studio Max for the plug-in.

Pros:

1. Full source code for plug-in and run-time modules for Direct 3D and OpenGL.
2. High-quality polygon-reduction algorithm with great performance.
3. No per-copy royalties on game sales.

Cons:

1. Plug-in only comes for 3D Studio Max. Users of other packages must roll their own conversion programs from library included in SDK.
2. Toolkit cost is up-front although pretty reasonable.
3. Users will need to adapt their game engines to work with the multi-resolution meshes.

The Blobs Go Marching Two by Two

This may come as a shock to some, but the world is not made up of corridors composed of completely planar surfaces. We live in a wildly organic place. Hills roll, muscles bulge and fountains splash. The world around you is filled with organic shapes which cannot easily be created out of triangles.

In fact, many of these objects are not even just lying around looking all organic. They slop, splash, waddle, and plop about you all the time. Many shapes around you are even in motion. These objects change shape effortlessly as you game artists crumple under the pressure of having to model such phenomena. When was the last time you saw a nice splashing fountain in a game, anyway?

Animators have faced the challenge of visually creating the organic world we live in for some time now. To help them out, commercial modeling packages have provided the artist with tools for creating organic shapes. One of the methods for creating organic objects is through the use of blobby balls that can be combined together to form a clay-like sculpture. The commercial animation package developers have realized the usefulness of this technique and coined all sorts of proprietary terms for their version. You may have seen ads for meta-balls, meta-clay, blob-modeling, and various other ways of combining the term "meta" with some form of goop.

To create an object from this meta-goop, an artist drags around primitive elements, usually spheres, which represent the rough shape of the object. Each of these elements has a center position and several parameters associated with it. These parameters define how the element will interact with the particles and world surrounding it. You can see an example structure for a meta-goop particle in Listing 1.

The position describes the center of the element. I also need to keep track of the radius of influence of the element (actually squared so I save some math later) and the strength of the element. This strength parameter defines how the element will affect the space surrounding it.

The elements interact with each other by creating an energy field around them. This is similar to the way planets create a gravitational field for a solar system. It is possible to evaluate the energy of the system at an arbitrary point in space. The formula to determine the amount of energy that an element contributes to the point is given as:

```
distance = squaredDistance(&goop->
   position,&testPosition);
if (distance < goop->radiusSquared)
{
   falloff = 1.0f - (distance/goop->
      radiusSquared);
   fieldStrength += goop->strength * falloff
      * falloff;
}
```

By running this formula over all the elements in your system, you get the exact field strength for that position. The energy field creates some interesting data but is not much of an object. What I want to create are particles that will visually grow together as they get closer. You can see an example in Figure 1. In order to create an object that will show this visual aspect of the energy field, it is necessary to define a value that will represent the outer shell of the object — the threshold.

The energy field varies in strength from zero on up at any position you may evaluate. In fact, there is nothing to keep you from defining negative strength for an element, creating negative regions, or holes, in the energy field. This is useful for effects such as denting and the like. To define the surface of the object in the field, I can set an arbitrary threshold giving the object its final shape.

The threshold value defines the boundary between the area inside and

LISTING 1. *A meta-goop particle.*

```
typedel tMetaGoop
{
   tVector   position;
   float     radiusSquared;
   float     strength;
};
```

The "meta-goop" seen here produces results that are difficult to create with traditional modeling techniques.

When not splashing gloop around his kitchen floor, Jeff can be found creating real-time graphics applications at Darwin 3D. Fling some bits of your own his way at jeffl@darwin3d.com.

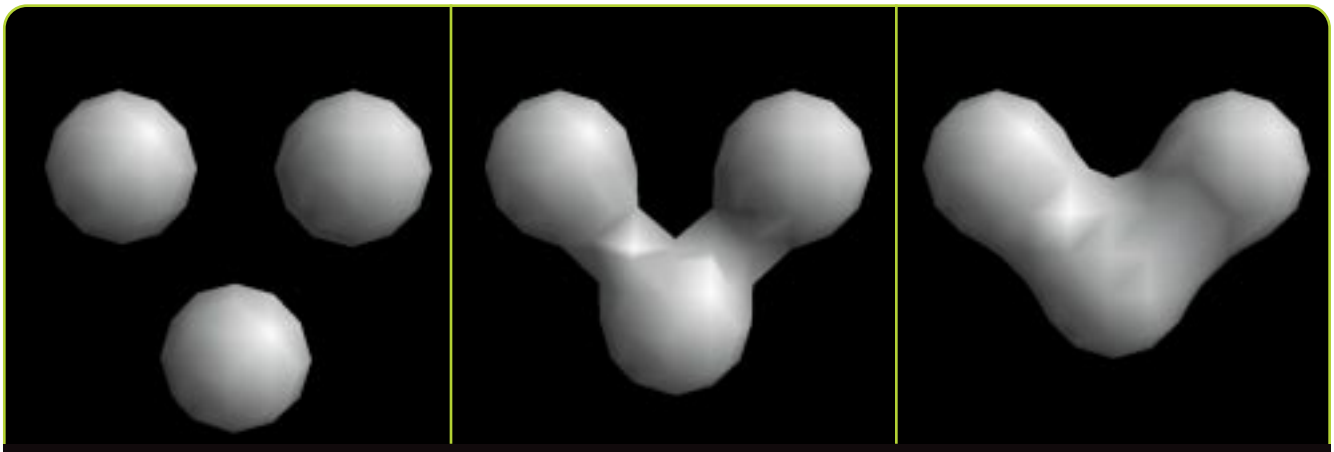


FIGURE 1. *Our goal is to make our three particles visually grow together as they get closer to one another.*

the area outside the shell of the object. Figure 2 shows how an example threshold value creates a boundary in a 2D energy field created by three meta-goop entities.

By adjusting this threshold value for the energy field, as well as adjusting the strength, position, and effective radius of individual entities, a great variety of objects can be created. But I still need to talk about how.

Walking on Eggshells

By creating a few meta-goop particles and setting some values for them, I have created my meta-goop system. Run that goop through a function that evaluates the energy field, apply a surface threshold, and I have the surface shell for the meta-goop object defined. But the problem remains, how do I draw it?

I could step across the entire 3D space defined by entity radii and evaluate the field. Anywhere the returned value is equal to the threshold, I could draw a solid cube the size of the steps taken. This sounds pretty good. Sounds like it would work. Actually, it sounds kind of familiar. It sounds kind of like volume rendering of voxels for applications such as viewing CAT scan data. In fact, that is exactly

what I would be doing if I took this approach.

However, rendering the energy field this way can lead to pretty chunky looking images unless the step size is fairly small. This is because the energy field is continuous over the entire range of the model world. However, the steps I took walking across the field are in discrete steps. If the steps are too big, the image can look chunky. This is analogous to drawing a line on a computer graphics screen. If the resolution of the screen is too low, the line can look very jagged. This unfortunate condition is known as "the jaggies" and requires some form of smoothing or antialiasing to make the lines look better.

Unfortunately, decreasing the step size in my energy field will greatly increase the amount of calculations that must be made. Therefore, it is necessary to find a way to smooth out the voxel image — sort of antialias the meta-surface.

CAT Scans and Game Development

Fortunately for me, the graphic visualization and medical imaging industries have been dealing with this issue for quite some time. Wyvill and McPheeters in 1986 and Lorenson and Cline in 1987 independently developed a system called "marching cubes" which enables you to render a polygonal approximation of a voxel field. One possible unfortunate circumstance is that this algorithm may be tainted by a software patent and I am investigating how this will affect the issue (see Sidebar).

That aside, the way marching cubes works is pretty simple. Divide the region you wish to render into a regular 3D grid. Evaluate the energy field at each position on this grid. Now, consider the grid cube by cube. If the energy function at all eight corners of the cube are less than the threshold level, the entire cube is outside the meta-object and the cube can be ignored completely. Likewise, if the corners are all greater than the threshold, the cube is completely inside the object and can also be ignored. The only cubes that need further consideration are those that have corners both inside and outside the meta-object. These cubes are on the object surface and will be part of the final render.

The Marching Cubes Patent Question

As many of you who have met me and heard me rant on the topic know, I believe algorithmic software patents are totally wrong. I feel they completely halt continued development down interesting research pathways by shrouding a topic with legal pitfalls. Graphics researchers create progress by building on the work done by others before them. I like to imagine the state of the industry if Bresenham had patented his method for drawing a line on a graphic display and then charged a licensing fee for every line drawn.

The topic of volume rendering is an interesting case in point. As an obvious next step in the visualization of volume data, it was reported by researchers in several publications. However, General Electric apparently owns a patent on the technique via the Lorenson and Cline implementation (U.S. patent #4,710,876). As an actual apparatus to display medical imaging data, I can understand it. However, the patenting of a "method for displaying three-dimensional surface images" seems pretty broad to me.

I have been told by someone via e-mail that GE aggressively enforces this patent. However, it is not clear to me how this would apply to the rendering of an isosurface in a game. Does this mean that any modeling program using these techniques must pay a license to GE? If I create a game using a derivative of marching cubes and it is a big hit, am I going to receive a stealth patent letter in the mail demanding a percentage? How derivative does it need to be? The prior art on this topic seems limitless, but what can I use as a reference and still be safe?

With the record number of software patents being filed, this is going to become an increasingly difficult issue for game developers in the future. I am actively researching the issue and hope to report on the results in a later column. Anyone with information on the topic, please let me know. In the meantime, always document your research from public journals as best you can. Ignorance is not bliss in this situation.

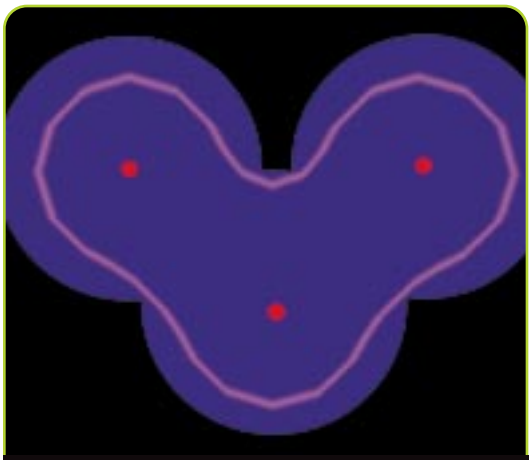


FIGURE 2. *Creating a boundary threshold in a 2D energy field.*

LISTING 2. *Finding the intersection point.*

```
void FindIntersection(  tVector *a, tVector *b,
              float aVal, float bVal,
              float thresh, tVector *result)
{
/// Local Variables ///////////////////////////////////////////////////////
tVector diff;
   float  ratio;
///////////////////////////////////////////////////////////////////////////////
   VectorSubtract( a, b, &diff);
   ratio = (thresh - aVal) / (bVal - aVal);
   VectorMultiply( &diff, ratio);
   VectorSubtract(a,&diff,result);
   if (aVal > bVal)
}
```

FIGURE 3A. *One vertex inside and the rest outside create one triangle.*

FIGURE 3B. *One vertex outside and the rest inside also make one triangle.*

FIGURE 3C. *Two vertices outside and two inside create two tringles.*

A cube has eight vertices. That means that there are 256 possible combinations of how a surface can intersect with the cube. If you consider symmetry, the number of possibilities reduces to 14. Much of the literature on surface generation using the marching cubes routine deals with optimizing for those 14 special cases.

However, there is an easier way I have seen termed "marching pyramid." If you consider a cube of eight vertices as being composed of five tetrahedrons with four vertices each, the problem is greatly simplified. There are now only three very simple cases to deal with. The cases are the following:

1. One vertex is inside the surface and the rest outside.
2. One vertex outside the surface and the rest inside.
3. Two vertices outside and two inside.

That is all I need to consider. In cases 1 and 2, a single triangle is generated. In case 3, two triangles are generated. You can see the three cases represented in Figures 3a–c.

Once the vertices of the pyramid are classified, the actual vertex positions of the triangles created are obtained by linear interpolation of the corner values along each edge. You can see the code for this in Listing 2. As there are five tetrahedrons making up each cube, the number of triangles generated with the marching pyramid technique is greater than what would be created from simple marching cubes. However, the classification and creation step is much simpler and the resultant surface is a more accurate approximation of the surface. On current 3D graphics hardware, the extra triangles shouldn't affect performance too much.

Goopy Games

I hope it is now clear that these meta-goop techniques can be used to create interesting organic objects suitable for real-time display. However, there are several aspects that actually make them ideal for use in games. For one, they are procedurally created. Complex structures can be generated from simple data structures consisting of the location and attributes of each particle in the system. There is no need to store a complete mesh.

In addition, the meta-object can be tessellated to different levels depending on the initial grid size of the voxel space. This gives the game a dynamic level-of-detail component that is needed in these days of varying hardware performance.

You can attempt generation of the objects in real time through efficient optimization of the surface approximation routine. You could also simply decide to create the objects at load time and display them as traditional polygonal objects during the actual game, or evaluate the mesh only when the state of the goop elements changes. This kind of flexibility makes for easy integration into a variety of applications.

I didn't even discuss how the surfaces could be rendered. One obvious choice would be to apply environment-mapping techniques to create the chrome creature from *Terminator 2*. Likewise, you could apply bump-mapping techniques to bring a water creature to life. I think an interesting application would be to combine meta-surface techniques to a particle system like the one I described last summer ("Spray in Your Face," Graphic Content, July 1998).

For more fun, get my demo application off the *Game Developer* web site (http://www.gdmag.com). This will allow you to play with the creation of meta-goop and start spreading some slop around your games. ■

FOR FURTHER INFO

- Greene, Ned. "Voxel Space Automata: Modeling with Stochastic Growth Processes in Voxel Space (Proceedings of Siggraph 89)." *Computer Graphics*, Vol. 23, No. 4 (Aug. 1989): pp. 175–184.
- Lorensen, William, and Harvey Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm (Proceedings of Siggraph 87)." *Computer Graphics* Vol. 21, No. 4 (Aug. 1987): pp. 163–169.
- Watt, Alan, and Mark Watt. *Advanced Animation and Rendering Techniques*. Reading, Mass: Addison-Wesley, 1993.
- Wyvill, Geoff, Craig McPheeters, and Brian Wyvill. "Data Structure for Soft Objects." *The Visual Computer* Vol. 2, No. 4 (Aug. 1986): pp. 227–234.

Web Resources

- http://www.students.cs.ruu.nl/people/jedik/Methods/Surface_fitting/Marching_cubes.htm
- http://www.swin.edu.au/astronomy/pbourke/modelling

Mo-Cap and Keyframing, Sittin' in a Tree

As an artist, you've heard at one time or another some pretty common arguments: photo reference vs. "from memory," tracing vs. hand-drawing, or scanned images vs. hand-painted ones done in Photoshop. However, for animators working on computer games today, there's a new debate when it comes to character animation: keyframing vs. motion capture.

Analogous to the classic Luddite/Promethean struggle that occurs still in the scientific community, animators tend to divide themselves into two camps. On one side you have the purists who believe mo-cap is the evil product of technology run by marketing blowhards. On the other you have the strategist artist who is constantly looking for the better, faster way to get the job done. Because in the end if your artwork is featured in a computer game and not a gallery in New York City, it boils down simply to getting the job done.

Mo-cap can and will help get the job done faster and better. Like any other tool available to the computer artist/animator today, it can be used in various ways to various degrees. What I'm going to do is demonstrate a situation in which motion capture and keyframing can be married — no, *have* to be married together — to form a solution for one instance of character animation.

I've Got My Eye on You

Meet Orbb. He's cool. He's a character we created for QUAKE 3: ARENA (with textures by Kenneth Scott) solely for the weirdness of making him run around on his hands (Figure 1). Because of the animation system of the game, affording or even showing off any type of complex finger animations is impossible. However, using a little bit of problem solving Orbb becomes a great experiment in evenly combining motion capture and keyframed animation.

I'll explain. Figure 2 shows a more orthographic view of our ocular, podiatrically-challenged little friend. In the Q3A game engine, each character must be divided into three distinct parts: head, torso and legs. The parts are tied together using a simple triangle tag system (match tab *a* to tab *b*). The head and torso basically move around when you move your mouse around (free look) with the head motions slightly leading those of the torso. The legs are purely locomotive and couldn't care less what the upper body does. Of course, death animations are all-body inclusive.

So in Orbb's case his hands have essentially become his legs, his body casing became his torso and his eyeball became his head. Setting up and attaching him to a biped in Character Studio turns out looking something like what we have in Figure 3.

Notice that I didn't give him arms and that the head and torso aren't linked to an

FIGURE 1. *Q3A's Orbb marches to the beat of a different drummer.*

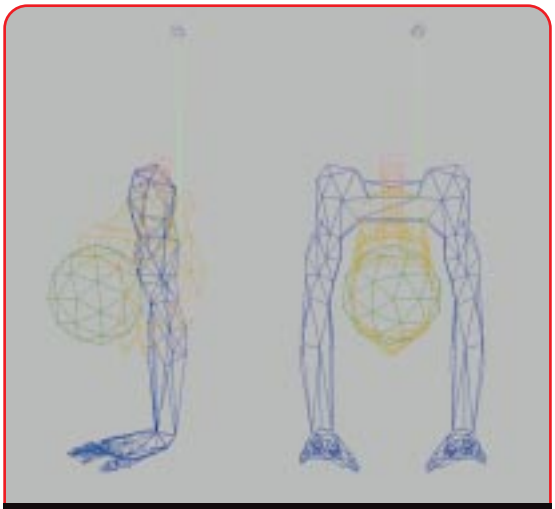


FIGURE 2. *An orthographic view of Orbb's peculiar anatomy.*

Still a form of computer game artist concentrate (just add imported beer) nearly 35 years in the making, Paul Steed (simply labeled "Steed" for easier marketability) will hopefully be applied to a new project by the time you read this. As usual, product information can be attained by dropping a line to psteed@idsoftware.com.

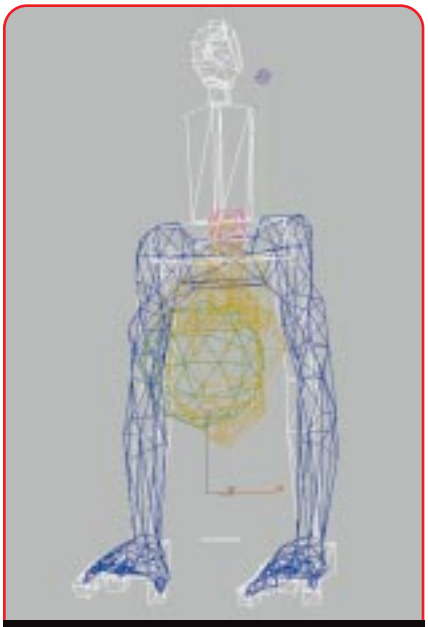


FIGURE 3. *Orbb has been attached to a regular biped skeleton.*

FIGURE 4. *Orbb's run cycle still needs work after plugging in the mo-cap data.*

FIGURE 6. *Adding the mo-cap data to the reversed legs gives better results.*

underlying skeleton. The reason for this is that I intend to apply mo-cap data in the form of death animations and locomotive animations mainly to the legs (arms), but I'm going to keyframe the fingers (toes), body and head animations. I won't even concern myself with what the unused skeletal parts of the biped do at this point since, given Orbb's unique anatomy, they don't matter (and I can't delete them).

So, taking a motion-captured run cycle I plug it into the character's skeleton (Figure 4). Something doesn't quite look right here. If indeed those are supposed to be arms, the elbows aren't quite bending right, now are they? And those toes sure don't look like prehensile digits. So my dilemma is how to make the legs act like legs while bending like arms. I wonder if I can turn the leg around in the up axis? First I'll detach the mesh from the skeleton, then select the upper thigh and rotate it so the knee faces the opposite direction (Figure 5).

Cool. Now we can apply the same mo-cap run data to the now-reversed legs and see what it looks like (Figure 6). This looks better. Next we reattach the mesh to the skeleton and keyframe those fingers...er, toes, doing something cool like pushing off, flicking out as they push off, and so on. Doing this keyframe work on the hands results in poses like the ones shown in Figure 7.

In adding keyframes to the fingers I try to exaggerate the motions a little. Just as a stage actor wears tons of makeup in order to be visible from the back row, amplifying animations is just as important in real-time games. Characters running around in a game like Q3A usually don't appear very large as you're trying to stuff rockets down their throat from afar.

So after applying the mo-cap data to the arms and keyframing the hands and fingers, Orbb runs like he's supposed to (Figure 8). I basically get to utilize all my animation files such as deaths, jumps, backpedals, walks, shuffles, or whatever, for Orbb's animation set as it pertains to his legs and center of gravity. I have to make some gross adjustments to make sure his weight distribution appears correct, but overall

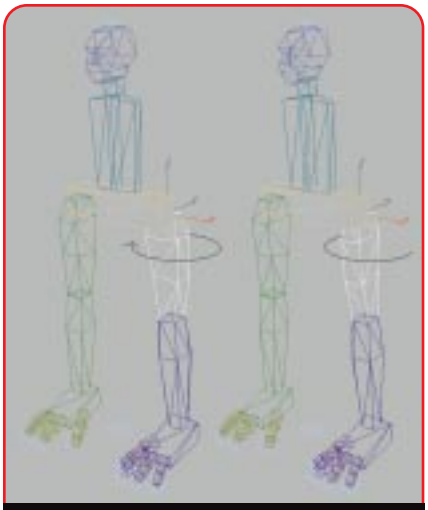


FIGURE 5. *Rotating the legs will make them act more like arms.*

FIGURE 7. *Keyframing offers us the precision to devise some cool hand poses.*

FIGURE 8. *Mo-cap and keyframe data combined to produce our finished run cycle.*

they work out pretty well. Once the locomotive animations are in, I keyframe the head and body casing to react accordingly.

This is just one of many possible situations that can support both motion capture and keyframing. However you look at it, mo-cap is just as useful as keyframing if you have a basic understanding of character animation and experience with keyframes. Rather than turning your nose up at an animation aid such as mo-cap, it behooves you at least to explore the potential benefit of the technology. After all, it's just...

...A Tool Like Any Other

A long time ago when I first started weight-training, I voiced my frustration at being so weak. This old bald guy from the monastery on base who was spotting me said, "Always remember, Grasshopper, the weights are a tool, not a measurement." The same applies to the methods and tools we use to create character animation with the computer. The character animator today no more becomes a talentless hack because he uses mo-cap than a master illustrator shows his inadequacies because he uses photographic reference for his painting.

I did my first animations back in high school when I'd have little stick people dancing along the edges of my textbooks running, fighting, somersaulting or just plain being lewd. Then I got into comic books and began telling stories with splash pages and sequential story art (panels) à la Kirby, Buscema, Byrne, Golden and Miller. Comic art or comic strips are the most basic example of keyframes. While not as literal as flipping a page and watching a little stick man come to life, each panel in a superhero comic is a static representation of a continuing dialog and dynamic flow for which your brain (instead of the computer) provides the "tween" frames.

Learning to draw and portray characters in this fashion is perfect basic training for character animation, since it forces you to see things in your mind clearly enough to put them down on paper. This translation of thought to media and, more importantly, the ability to recognize a *successful* translation is the key to better character animation because it gives you "the eye."

Simply put, having the eye means you can look at something and see it to be either one of two ways: right or wrong. If it's right then you can move on to the next task. If it's wrong, you keep working on it until you make it right. So many times someone shows me something or e-mails me some animation sequence and asks my opinion. When it's so bad that I don't know where to start with a meaningful critique I simply ask, "Does it look right to you?"

QUAKE 3: ARENA marks the first time I've personally dealt with mo-cap outside of the annual obligatory dog-and-pony shows at Siggraph and other trade events. I decided to try the mo-cap route because it's dead easy in Character Studio and for the simple reason our frame rate went from 10 FPS in QUAKE 2 to 15–25 FPS (depending on the animation). Turning to mo-cap makes sense as frame rates increase, since keyframing the subtle and nearly imperceptible nuances of humanoid character animation, if not difficult, is at least time-consuming. The difference between a six-frame run cycle and a 13-frame run cycle is obvious to say the least. Another reason is that my fellow animator Kevin Cloud decided to concentrate on other art aspects and leave me to do all the models and animations (nice of him, wasn't it?), so finding ways to save time became a priority.

However, to say that I find myself forgetting how to keyframe because I've implemented mo-cap into the workflow is just plain ludicrous. Mo-cap is a start and, if anything, a rough timing guide to assist your keyframing. I have yet to implement any motion-captured animation without at least some keyframed adjustment. This is not a problem for me.

The Mo-Cap Experience

The first session I did was with Greg Pyros and his crew at Pyros Pictures. An amazing martial artist and actor, T.J. Storm, and a fellow actress, Bobbi, were the talent for my first run at making animations for the characters in Q3A. I learned a lot from the session, but since it was my first time I came up a little short in preplanning and accurately predicting the implementation of the data in my animations. I also didn't know Character Studio very well at the time and failed to exploit its strengths and allow for its shortcomings. Knowing your pipeline and knowing all your software is crucial to successful mo-cap implementations.

When I did my next session I went to House of Moves. The crew at HOM definitely know their stuff and they were great to work with. I was quickly allowed to review the motions captured in crude wireframe playback to see if it was what I wanted. HOM also gave me videotape that matched the captured moves for both review and reference. There were differences dealing with House of Moves, but the most overwhelming difference was that I suited up to do most of the motions myself.

I consider myself to be in reasonably good shape so I wasn't afraid of the physicality of the shoot. Since I'm a ham at heart I knew I could act well enough to get reasonably expressive motions based on what I'd seen T.J. do at the earlier shoot, and more importantly, to get what I wanted from the characters I had already created. However, I fully realize that I'm in a unique position to have more freedom than most artists at most companies. It's great being the modeler, animator, mo-cap director and mo-cap actor. I get to make sure I get exactly what I want and need to get the animations right. I enjoyed the HOM shoot so much that when I did my third session just recently I decided to be the "talent" once again.

This time however, I decided to try yet another company a whole lot closer to home. Located in the idyllic, rolling hills and fauna of Wimberly, Tex., near Austin, Kei and the crew at Locomotion Studios provided a comfortable, relaxed and professional atmosphere (except they didn't have any beer stocked).

The animations I went for this time were more scripted than the animations I did at the other two studios, so it was more important for me to give a better performance. Instead being pieced together and used by the characters during real-time game play, these animations would be plugged into the characters mostly as-is to be used for rendered cutscenes. This is a great example of mo-cap saving you time on a project since the amount of work it

would take to make the subtle, expressive, realistic nuances I wanted in the animations would have taken far longer than I have to do them. Even in a worst-case scenario where the animation is too wooden, keyframe augmentation can still save the day utilizing accurate timing if nothing else.

In addition to offering immediate playback of the captured animation to see if it was what I wanted, Locomotion took the process one step further by filming the captures in MPEG format, giving me a digital movie of the performance. This proved extremely convenient, more so than videotape.

Extremely accommodating like the other studios, they were tolerant when I got wacky. For example, one thing I did for a particular character with a three-jointed leg was to walk backwards in order to give it a creepy quality when it walked forward. To support this, the guys at Locomotion quickly and easily played back the animation in reverse, which allowed me to adjust my performance until I got it right. Oh, and do you want to know the best thing about the session at Locomotion? Soft reflector balls. Trust me, when you're covered with the hard ones, your potential for lots of pain while doing mo-cap is high.

Just Where Do I Go for Mo-Cap?

As I've said, time-saving is another attractive quality of motion capture, but one that gets argued by artists because of the heavy cleanup required. That's what mo-cap service houses such as Pyros Pictures, House of Moves and Locomotion are for. You can let them do the clean up instead. By the time you've made the space, spent the money and trained the artists to support your own motion capture facility, the overall cost will be hard to beat by simply going to an expert. Having dealt with three of these companies that provide such a service in the past year has given me a fast and comprehensive understanding of the motion capture process. Understanding this process from start to finish is a necessity if you plan on using mo-cap.

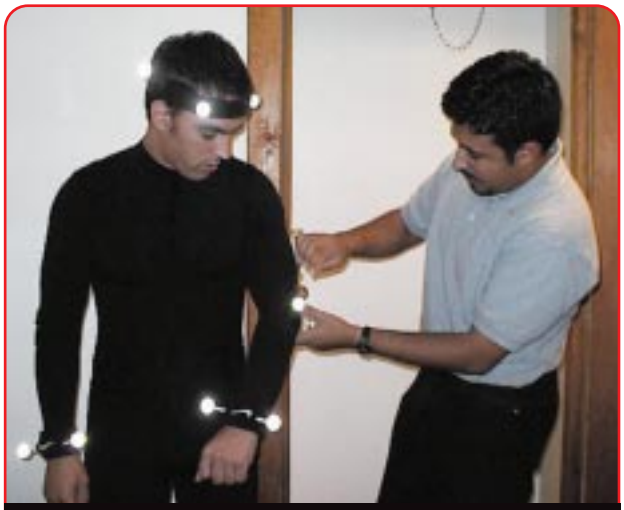
Therefore, consider the following as you tread down the motion-capture studio path. It could mean the difference between success or failure:

PREPLAN. By preplanning I mean storyboard, either literally or in your mind, each move you plan on getting. Ensure there are one or two people who track and manage the process from start to finish (you, the artist, being one of them). Also, come up with good file names for your motions. I usually stick to a six-character naming convention because it leaves room for different versions (takes) later on. Calculate the amount of time in seconds you estimate each motion will last and then a total for the shoot. Use that number of finished seconds as a starting point when you negotiate a price.

GET A BID. Once your list is formed, start shopping around. Just be sure you have a very clear idea of what you want before presenting it to a mo-cap studio. Don't be afraid to start a bidding war. These guys know how important patronage and word of mouth are in our industry.

GET GOOD TALENT. Although I really enjoy doing my own capture sessions, next

time I'll go back to paying someone else to hit the ground and writhe around in real...er, *mock* pain. I've found the key to getting a good performance is to find actors who are almost mime-like in their expressions and motions. Also remind your actor to go full speed and not slow down during performance. This may not be something readily apparent, but overacting or being overly conscious of devices tracking your every movement can be very obvious in the finished data. I've had to cut up to ten percent of the keyframes in any given mo-cap file due to the actor being too measured in his or her movements, or make substantial tweaks because hands were used to break a fall (or to remove other glaring forms of anticipation). If you have the inclination and the ability, I highly recommend doing your own motions at least once. It makes you appreciate what you ask the talent to do in all those hard reflector balls.



Steed knows the importance of wearing many hats during the motion-capture process, including this one covered with reflector balls at Texas's Locomotion Studios.

GET THE MOTION YOU WANT. Not much else I can say here. You're the client. You're plopping down the cash (usually 30 percent or so up front when you show up at the studio). You decide when the captured motion is right. Insist on a video or MPEG of the performance to review and just have around. If anything, it's something to show your boss and co-workers.

CHOOSE YOUR IN AND OUT TIMES. Basically, the in and out times are the points at which you want the motions you see on video to start and stop recording. This is obviously just so the mo-cap studio can calculate your total number of seconds captured, giving you exactly what you want when it happens. But never underestimate that little hitch or movement right at the beginning or end of a motion. While it's prudent not to pay for useless seconds of idle movement, you might be able to take a part of one motion and use it in combination with keyframes somewhere else.

IMPLEMENT THE CLEANED DATA. Once the data is delivered, plug it into your character and see how it works. I did captures for a character and decided I didn't like the implementation and just cut it from the list. Occasionally, some things really do just look better on paper.

PAY THE GUY. Guess this is the bottom line, isn't it? Rates vary, but if you shop around, like I've already suggested, you'll get the fairest price. The price of motion capture seems to keep dropping every time I get it done. This trend will no doubt continue as technology advances.

Speaking of cost, it is something to consider. For a small developer, using a mo-cap studio may be cost-prohibitive, but I'd encourage you to check it out anyway. All the people I've worked with have been extremely accommodating and flexible when trying to get my business. If you preplan appropriately and know exactly what kind of motions you want, the end cost will reflect your degree of organization.

Another key to making a mo-cap session successful is to involve the animator who will be using the data. Sure, those clipboard-carrying, coffee-cup-toting producer types have their uses and can baby-sit the fiscally irresponsible artist if they like, but at least one of the artists who will be working with the fruit of the session needs to be present during the shoot.

Can't We All Just Get Along?

Not too long ago, I read a couple negative commentaries on motion capture in a telephone-book-sized tome dedicated solely to character animation. This was the first time I ever heard mo-cap referred to as "Satan's Rotoscope." What a load of crap.

I hear statements like this and I think of that *e*-word — ego. Teams of people are responsible for games today, not individuals. Not to explore the possibilities and benefits of a new technology because you want the pleasure of proclaiming you did it all from scratch is part of the inanity that results in very late products being shipped. Sure, given all the time in the world, any animator worth his salt can create convincing and supremely realistic character animation. Well, we all know how much time we have to do our animations, don't we?

But, what the hey. Let's just say for a second that mo-cap *is* evil. The bane of the *real* artiste. If that's the case, then why not just toss that computer altogether and go back to traditional cel art animation and just scan the art in? Oh, but that would mean you'd have to use a scanner. Since that evil contraption digitally captures images that could be used in game art, such as texture maps, we should probably get rid of that bit of demonic technology as well. While you're at it, what about digital cameras, model digitizers, heck...even photographic reference of any sort? Maybe all example of modern techniques or technologies that assist the artist in his ability to meet his deadlines with work that (gasp) is sometimes better than he could have achieved on his own should be consigned as products of the nether realm.

The purists would secretly be not too unhappy about this because then they could weed out the real artists from those pretentious wannabe keyboard jocks. Then they could beat their chests in pride and reaffirm their prodigious training and stature.

Technophobia has no place in today's world of computer game development. Motion capture is here to stay and will antiquate keyframing about as quickly as electronic documents have turned our work environment into a "paperless" office. In the right hands, mo-cap is the perfect way to aid and enhance the keyframe animator, not replace him. ■

A Tale of Two Channels

PC gaming appears to be the technological peak of game development. However, the advent of Sega's Dreamcast, Sony's marketing of Playstation 2 technology, and the intriguing prospects of Nintendo's Dolphin system have helped to promote greater anticipation for the console.

Even Nintendo's Game Boy has shown phenomenal growth in 1999 and is attracting new titles and developers to the color version, while the next-generation, 32-bit Game Boy "Advance" is slated for release late next year. All these comparisons of platform technologies aside, the biggest obstacle to success in the PC gaming market is the structure of the PC business, which is built around the sales of PC hardware to businesses and education outlets. Games may be ubiquitous, but the PC industry still sees the game industry as a marginal interest, and sometimes solely as a means to sell the latest CPU. Examining the market of console game users and the channels for console products is a sobering lesson in how far the PC industry has to go to become consumer-savvy.

The PC game business has always been hampered by a lack of shelf space and sufficient sales outlets. This situation is unlikely to be resolved while the market remains driven by the upgrade cycles of Microsoft and Intel. In the meantime, the console gaming market is benefiting from penetration into both traditional retailing channels and existing PC sales channels.

Channels Young and Old

While online sales of PC products is considered to be a hot area of growth, the vast majority of sales still come from third-party retailers and distributors who rely on their expertise with applications to serve a desired market. It's rare to find a reseller or system integrator that is focused on games. In contrast, console products are predominantly retail-based and require no third-party support or intervention. For PCs, there are more than 100,000 companies in the U.S. alone that act as middlemen for computer manufacturers, and who supply systems, upgrades and services to users. In such a large, competitive market, specializing in hardware sales to game players would make little sense, and trying to compete in the arena of PC gaming software is unlikely to enliven any reseller's bottom line.

By contrast, a fully configured network of systems for a small business or corporate customer has potential for lucrative service and support contracts, or may be solely beneficial in selling a reseller's software expertise. So, while the $125 billion in sales that the PC channels produce is an impressive number, it's spread across a vast mix of dealers and resellers. These consist of value-added resellers (VARs), retailers, distributors, systems integrators (SIs), mom-and-pop stores, and everything from a single-person consultancy to big service companies such as EDS. Stacked up against these hardware-driven distribution channels is a console business driven by cute game characters and mass-market promotions.

Simply put, PC channels are relatively young compared to the retail channels that console games have penetrated. PC channels are as old as the PC, while retail channels date back to before the electronic age. This points out the gaping chasm of consumer savvy between traditional PC sales channels, and the retailers that thrive on N64, PSX, and Game Boy products. Furthermore, the next generation of console products promises to constrict the PC gaming market by providing powerful platforms with multimedia and online capabilities. Next-generation consoles may not be PCs, but they don't aim to sit on someone's office LAN, either.

The Console Player's Choice

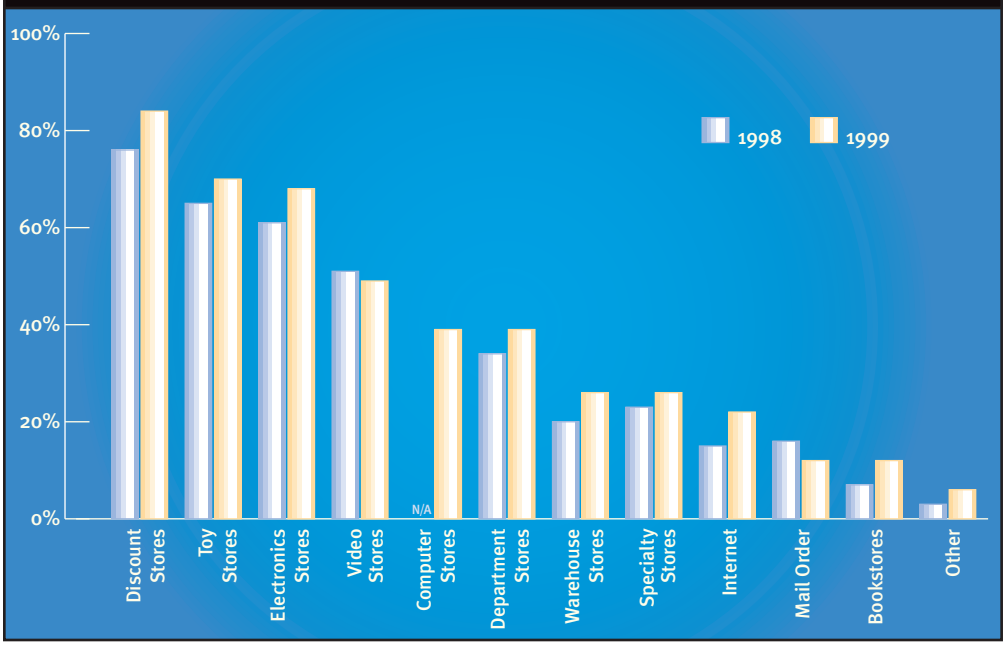
At the third annual Electronic Gaming Summit this past August, Ziff-Davis announced the results of the Video Gaming in America research report. This report found that purchase impact is influenced more by brand power of a character or game than by the knowledge of the publisher or developer. Surprisingly, the buying habits of game players, according to the report's findings shown in Chart 1, point out their preference for discount stores where impulse buying and pricing play a key role. Console games seem to have better branding and get into the places where the most buying occurs. This is despite the fact that at places like Wal-Mart, PC games have to be priced below $20 to qualify for significant sales. If that weren't bad enough, the reputation of consoles as high-technology products, which is reflected in the sales of console games through computer superstores.

The other advantage that the console market has is rentals. "Core" game players (defined in the report as those consumers who purchase two or more games per month) are averaging more than three rentals per month, while "casual" players (defined as those who purchase fewer than two games per month and play fewer than four times

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

per week) average about 1.4 rentals per month. The real kicker is purchase rates after rental are 84 percent and 65 percent respectively for core and casual players. Video rental outlets can devote anywhere from 10 to 20 percent of their shelf space to games, but generate in excess of those figures in percentage of revenues from rentals. There is really no mechanism in place, or likely to be put in place, to create the same rental opportunity for PCs. The PC always has the benefit of the limited demo software package bundled with game magazines and available for download, but these options pale in comparison to being able to take a game home to play in full for three days.

CHART 1. *Game players are shopping across all outlets (source: Ziff-Davis).*



Another interesting finding in the report is the influences on players' purchasing choices across core players, casual players, and "average" players (defined as consumers who purchase fewer than two games per month but play four or more times per week), shown in Chart 2. Friends, television, and in-store game demos played a key role here. In-store demos of PC games are unlikely to happen anytime soon, unless you have a publisher with a very significant marketing budget to spend, or PCs end up costing $100, and you can load 10 games within 60 seconds.

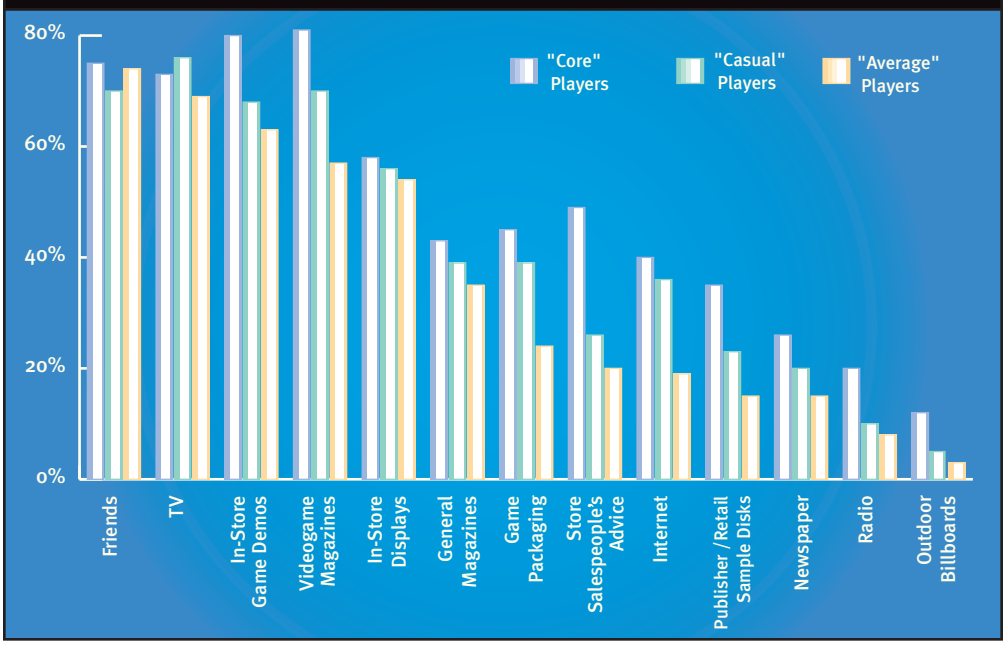
The PC Branding Problem

PC games, with some exceptions, tend to be more sophisticated simulations and immersive environments. As a result, PC games lack some of the "cute" character brands of console games. At a more specialist level, the PC has programmers like John Carmack and designers like Sid Meier, but the console industry has Shigeru Miyamoto and David Perry.

To reconcile these two worlds, PC game channels have to get closer to the console market in terms of marketing sophistication and audience appeal. PC gaming will not ultimately die, but it will become cornered, and that will affect the flow of new talent into the industry. That in turn would affect innovations in technology and titles. It looks like the console market has finally grown up, and now it's time for the PC market to do the same. But it won't happen until the gaming industry figures out how to break the infrastructure of PC sales channels.

Recently, the Good Guys chain of electronics stores announced it was ceasing PC sales, CompUSA said it would close some of its stores, and OfficeMax has felt the drain of lower PC prices and profits. Meanwhile, displays of console games and hardware in CompUSA stores continue to grow. ■

CHART 2. *Consumers' sources for videogame purchasing information (source: Ziff-Davis).*



P i g C ed S face W k he Ni e d 64

by Mark A. DeLoura

ame console programming is largely a secret art. The technology and APIs are kept hidden by nondisclosure agreements, and you won't find development kits for game consoles at your local software store. As a result, programming for game consoles is something you just don't hear much about.

While specific techniques for programming Nintendo's current game console are well-known within that particular developer community, they are virtually unknown among PC developers, or developers looking to do cross-platform titles. This article will give you some insight into the inner workings of the Nintendo 64 (N64). Much of what I'll discuss in this article hasn't even been released to authorized N64 developers. Nintendo has chosen to

Mark A. DeLoura (markde01@noa.nintendo.com) is the software engineering lead for the Product Support Group at Nintendo. He's been working on the Nintendo 64 since the first hardware dev kits showed up, and he damn near cried the first time he booted up SUPER MARIO 64. Now he's working on high-tech wizardry for Nintendo's next-generation console, Dolphin.

pull back the covers to help developers squeeze the last ounce of performance out of the machine. We hope that this article will help N64 developers do just that, and encourage other developers to explore N64 programming.

After a quick discussion of the N64 architecture, we'll dig down deep and design some custom Reality Signal Processor (RSP) microcode, which tessellates a Bézier surface as shown in Figure 1. The RSP is a very powerful custom chip in the N64, and until now the details of programming this chip have been kept secret. In a sense, we at Nintendo have decided to let the cat out of the bag. You'll get a feel for the incredible power of this chip and see why N64 is capable of great 3D graphics with features that still aren't available in consumer 3D cards.

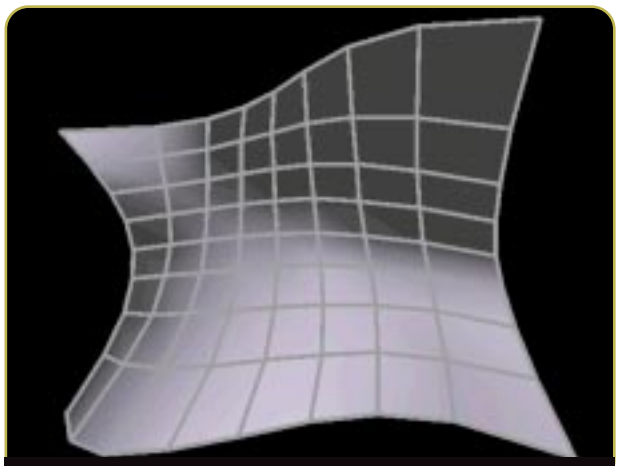


FIGURE 1. *This Bézier surface has been tessellated by the microcode we develop in this article, and rendered by a Nintendo 64.*

Nintendo 64 Architecture

The Nintendo 64 is designed around two main processing components (Figure 2). These two elements are a MIPS R4300i CPU, and the Reality Co-Processor (RCP), which is a custom chip. The simplicity of this architecture makes N64 programming very straightforward. In addition to these processors, the N64 contains 4MB of Rambus DRAM (RDRAM), four controller ports, and a cartridge port. The memory is expandable and a 4MB Expansion Pak is currently available.

The N64's custom RCP runs at 62.5MHz. It is primarily composed of two parts: the Reality Signal Processor (RSP) and the Reality Display Processor (RDP). The RSP processes display lists which are sent from the CPU. It performs all matrix and vertex computations and outputs triangle commands to the RDP. The RDP takes this information, loads the texture cache from RDRAM, and renders fully MIP-mapped, anti-aliased, Z-buffered triangles to the frame buffer. This design leaves the CPU free to perform physics calculations, advanced artificial intelligence, sound processing, and other game functions.

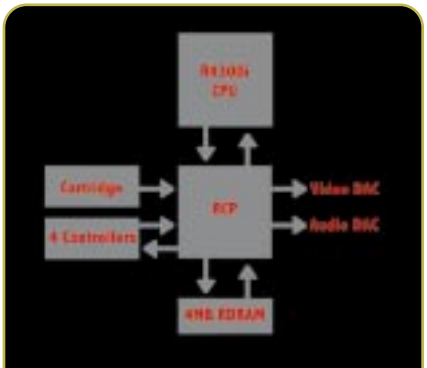


FIGURE 2. *The Nintendo 64 architecture is simple and elegant.*

RSP Architecture

The RSP is modeled on a general-purpose 32-bit RISC processor. It includes 4KB of memory for code (IMEM) and 4KB of memory for data (DMEM). Programs which execute on the RSP are known as microcode. Nintendo provides a standard suite of microcode to all N64 developers, including 3D transformation and lighting code, line-drawing code, sprite routines, and audio processing. Due to special features of the RSP, it is very well-suited for computationally heavy tasks such as 3D graphics calculation and audio mixing.

In addition to 32 32-bit scalar registers, the RSP includes 32 128-bit vector registers. These vector registers can be addressed in a variety of ways, but they are ideally used as eight shorts (also called vector slices). Each slice has a 48-bit accumulator associated with it that can be used to store intermediate results. Using the vector registers and accumulators, a vector operation can be performed which multiplies two vectors and adds the result to the current accumulators, giving 16 calculations in one cycle.

The RSP can actually execute a vector operation and a scalar operation each cycle. This means that it's possible to do 17 calculations per cycle. With carefully tuned microcode, it is possible to reach a maximum of just over one billion operations per second.

The Microcode

This high-speed programmable architecture was very forward-thinking at the time the Nintendo 64 was designed. It has enabled Nintendo to provide a set of standard microcode libraries which make 3D programming easier for the novice. At the same time, elite programmers are able to code up special routines which are optimized for their own games or enable unique functionality. During the life span of the N64, the 3D performance has nearly doubled as a result of microcode optimizations.

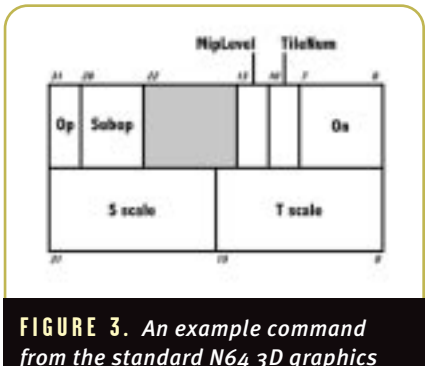


FIGURE 3. *An example command from the standard N64 3D graphics microcode. This command turns on texture mapping using a specific texture tile.*

Microcode Execution

First let's talk a little about the structure of microcode and how to use it. Microcode is exe-

cuted through the use of an RSP task. Tasks are command lists (graphics display lists or audio commands) which indicate a series of operations for the microcode to perform. They are executed in parallel with the CPU. In order to start an RSP task, you create the command list and pass it to the RSP along with pointers to the microcode and various buffers that the microcode needs. Then you call a simple function to start RSP execution and control is immediately returned to the main program while the RSP begins processing commands.

The RSP can communicate with the RDP or CPU during execution if necessary. For example, most versions of the 3D microcode communicate with the RDP, feeding it triangles and other data to render to the frame buffer. Other versions of microcode communicate with the CPU when data is ready. For example, the Z-Sort microcode can be set up to alert the CPU after a number of objects have been processed so that the CPU can work on these objects in parallel. When the RSP completes the task, it signals the CPU so that the user program can send the next RSP task or use this information for synchronization.

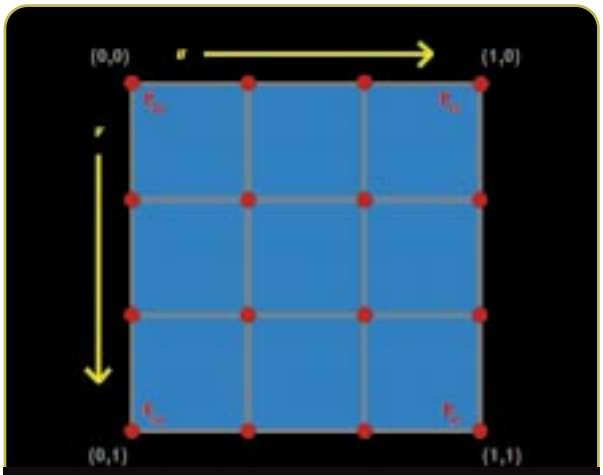


FIGURE 4. *Bézier surfaces are defined in a biparametric space. Sixteen control points are used to define the surface completely.*

The Command Loop

The microcode command loop sequentially goes through commands which have been DMA'd into DMEM from the command list. Similar to assembly language instructions, the commands have bitfields which indicate the RSP function desired. In the microcode command loop, the opcode and subopcode bitfields are masked off and used as an offset into the function jump tables (also stored in DMEM) to determine the IMEM function location.

In the standard graphics microcodes, each command is a 64-bit doubleword. The opcode and subopcode are contained in the upper bits, and lower bits are reserved for data being passed as function parameters as shown in the example in Figure 3. The data bitfields are masked off in the main loop and stored in separate registers before jumping to the function requested.

The DMA Engine

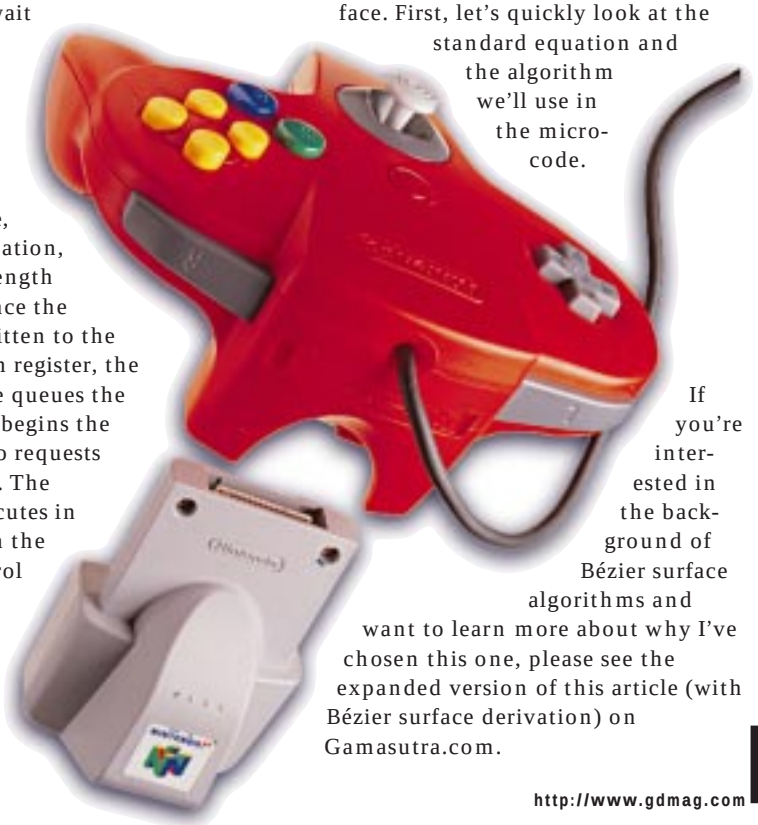
The RSP includes a set of registers which control the DMA engine. Since there may be multiple requests for DMA pending, the microcode must check the DMA Busy register before submitting its request. If a request is being processed and there is already another request pending, the microcode must wait before submitting a request. A request is made by altering the DMA Source, DMA Destination, and DMA Length registers. Once the length is written to the DMA Length register, the DMA engine queues the request and begins the transfer if no requests are pending. The transfer executes in parallel with the RSP so control is immediately returned to the microcode.

Using Curved Surfaces

With this basic understanding of the N64's workings behind us, let's move on to the main focus of this article, using curved surfaces. Curved surfaces are not supported in the standard N64 microcodes. But if you want to render curved surfaces, it makes a lot of sense to do the heavy computations required on the vector processor. Now, we're not actually going to render curved surfaces. We'll take a curved surface representation and tessellate the surface into polygons which the N64 then renders.

For our purposes here, we are going to use Bézier surfaces. A Bézier surface is a curved bicubic surface, similar to Hermite surfaces, B-spline surfaces, and NURBS. The Bézier is mathematically complex enough for us to be able to create interesting surfaces, while not being so difficult to compute that we're only going to be able to do a couple per frame. If you need to brush up on curved surface technology, check out the list of references at the end of this article.

There are a number of algorithms we could use to tessellate a Bézier surface. First, let's quickly look at the standard equation and the algorithm we'll use in the microcode.



If you're interested in the background of Bézier surface algorithms and want to learn more about why I've chosen this one, please see the expanded version of this article (with Bézier surface derivation) on Gamasutra.com.

Bézier Surface Equation

A Bézier surface is a parametric surface (u,v = [0,1], [0,1]) defined by its 16 control points p_{ij} which form a 4×4 grid, as shown in Figure 4. The common form for representing this surface is:

$$Q(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} p_{ij} B_i(u) B_j(v)$$

The functions $B_i(u)$ and $B_j(v)$ are the Bernstein polynomials which are also used for Bézier curves.

The edges of a Bézier surface are each Bézier curves. Since only the end control points of Bézier curves lie on the curve, we can extrapolate that the corner points of the surface are the only control points which lie on the surface. All twelve of the other control points influence the shape of the surface, but are not on the surface itself. For this article, we'll create a microcode that tessellates a Bézier surface into an 8×8 grid of quadrilaterals.

Tessellation by Evaluation

The most direct way to slice a Bézier surface into polygons is by calculating the above $Q(u,v)$ double summation on a regular grid. Performing this in a very optimized way, each surface vertex we calculate requires 54 additions and 108 multiplies. That's a lot of work to do when we're planning to create a 9×9 grid of vertices.

Central Differencing

The way we're going to generate points on the Bézier surface in this article is through the use of central differencing. Central differencing gives us an easy way to find the midpoint of a Bézier curve without having to keep track of control points for each subcurve. We can split the edge curves at their midpoints, and then split the surface across these midpoints to create four subsurfaces. This process can be repeated recursively to create an arbitrarily fine mesh. (For details on this algorithm please see the previously-mentioned article on Gamasutra.com, or Brian Sharp's series of articles on curved surfaces, June–July 1999.)

The central differencing algorithm has a hefty initialization cost due to the computation of second partial derivatives (Quu, Qvv, $Quuvv$) at each corner control point. But every curve subdivision after that will only cost us 18 additions and 18 multiplies. The memory footprint is 24 bytes per subdivision, and there are 77 subdivisions necessary to create our mesh. This will fit in our 4KB DMEM nicely.

Writing the Tessellation Microcode

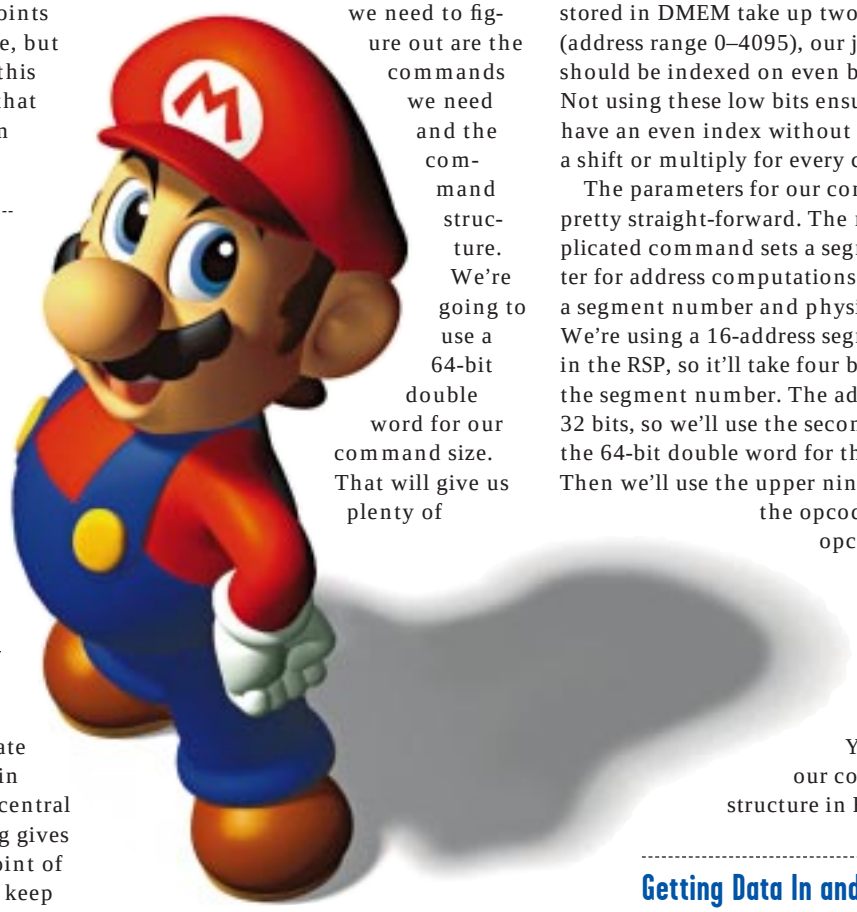
Now that we've chosen the algorithm to tessellate the surface, let's get back to work on the microcode itself. The first things we need to figure out are the commands we need and the command structure. We're going to use a 64-bit double word for our command size. That will give us plenty of room to store the data for each command inside the instruction. The commands and parameters necessary for our tessellation microcode are:

1. Set RSP segment (segment number, physical address).
2. Load control points (segment address).
3. Perform tessellation.
4. Save surface vertices (segment address).
5. End display list.

I'll describe these commands further in a moment.

Since we only have five commands, we can just use a 3-bit field for the opcode. Fortunately, the standard graphics microcodes all use a 3-bit opcode field and 6-bit subopcode field, so we'll use that. But we'll just wedge all our instructions into the subopcodes for one primary opcode. Then we can reuse a lot of the main command loop routines from the standard microcodes, including the display list DMA routine that loads commands into the DMEM buffer for us. The low bit of the opcode field and subopcode field are not used. Since microcode function addresses stored in DMEM take up two bytes (address range 0–4095), our jump table should be indexed on even bytes only. Not using these low bits ensures that we have an even index without performing a shift or multiply for every command.

The parameters for our commands are pretty straight-forward. The most complicated command sets a segment register for address computations. It requires a segment number and physical address. We're using a 16-address segment table in the RSP, so it'll take four bits to hold the segment number. The addresses are 32 bits, so we'll use the second half of the 64-bit double word for the address. Then we'll use the upper nine bits for the opcode and subopcode fields and follow it with four bits for the segment address. You can see our command structure in Figure 5.



Getting Data In and Out

Before we code up the tessellation algorithm, let's figure out how to get data in and out of DMEM. The "set RSP segment" command fills an entry of our 16-entry segment/offset table, which is stored in DMEM. This table makes some programming tasks easier, such as swapping the frame buffer each frame. The segment table stores 24-bit

FIGURE 5. *Our command structure is similar to the structure of standard N64 microcode commands. We use this structure for all of our commands.*

offsets which are added to any address sent to the RSP. The segment table index is stored in bits 24–27 of the addresses passed in. The low 24 bits of the segment address are added to the 24 bits stored in the segment table. Since our physical address range is 0–0x007fffff (8MB), 24 bits is enough.

Prior to tessellation we need to load the control points into DMEM. Our "load control points" command simply takes an address as a parameter. The address is passed to the segment address translation routine, which uses the segment table to convert the address to a physical address. The DMA engine is called to bring the 16 control points into DMEM from this physical address.

After tessellation, we need to save the surface vertices we've computed, using the "save surface vertices" command. We'll pass in an address and the segment address translation routine will convert it to a physical address. That physical address is used to program the DMA engine to copy our 81 surface vertices to RDRAM.

The "end display list" command simply flags the RSP to quit. It executes a break, which signals the CPU, and alerts our main program.

Data Formats

The first thing our "perform tessellation" command does is perform a simple translation from control point format to surface vertex format. So let's talk about these formats.

The Nintendo 64's standard vertex format uses 16-bit coordinate ranges, which are s15 quantities (one sign bit and 15 integer bits). This gives vertex coordinates an effective range of +/– 32KB. It makes sense for us to use this same format for control points, but since we're just tessellating, we really only need the point position. Rather than wasting the extra space for colors and texture coordinates when we DMA the control points into DMEM, our format will only represent the *x*, *y*, and *z* position as signed shorts.

The surface vertex format is more complicated. The central difference algorithm describes four sets of values that each vertex needs to track. These are:

1. $Q(u,v)$: Position
2. $Q_{uu}(u,v)$: Second partial derivative in *u* at this vertex.
3. $Q_{vv}(u,v)$: Second partial derivative in *v* at this vertex.
4. $Q_{uuvv}(u,v)$: Second partial derivative in *u* of the second partial derivative in *v* at this vertex.

All of these values are vectors of *x*, *y*, and *z*. Since the vector slice size of the RSP is 16 bits, and the control point coordinates are 16 bits, we're going to stick with 16 bits for these coordinate values as well. We'll have to tweak our math to minimize overflow and underflow, but it will pay off in performance.

One final note on formats. Each vector register contains eight vector slices. But each of our points contains three values (*x*, *y*, and *z*). We're really just going to make things confusing if we try to stuff two of three coordinates from one vertex into a vector register, along with two other vertices. So let's insert a junk (we'll call it *j*) field at the end of each of these vertices. This will also give us much better alignment in DMEM.

Now that we have our formats defined as in Listing 1, it's a simple task to convert from one to the other. Actually, all we need to do is copy the 64 bits from each corner control point (*x*, *y*, *z*, and *j*) into the beginning of each corner surface vertex.

Corner Initialization

Now we need to compute the second partial derivatives described above for each corner of the surface. Fortunately, the second partial in *u* and the second partial in *v* at each

LISTING 1. *Formats for data storage in DMEM. The* j *fields are unused, we include them for data alignment.*

```
struct ControlPoint {
        s15     x, y, z, j;
};

struct SurfaceVertex {
        s15     qx, qy, qz, qj;
        s15     quux, quuy, quuz, quuj;
        s15     qvvx, qvvy, qvvz, qvvj;
        s15     quuvvx, quuvvy, quuvvz, quuvvj;
};
```

LISTING 2. *Pseudocode for computing* $Q_{uu}(o,o)$ *and* $Q_{vv}(o,o)$ *using vector processing.*

```
# Load the vector registers with point data
vload     vectora, P[0,0], P[0,0]       # = x y z j, x y z j
vload     vectorb, P[1,0], P[0,1]
vload     vectorc, P[2,0], P[0,2]

# Do vector computations to simultaneously compute quu and qvv.
vadd              vectord, vectora, vectorc     # D = A+C
vmul              vectore, vectorb, vconst[5]   # E = B*(-2)
vadd              vinter, vectord, vectore      # inter = A-2B+C
vmul              v00, vinter, vconst[3]        # v00 = 6*(A-2B+C)
vstore2           v00, v00uu, v00vv             # Store results to uu and vv fields
```

corner control point can be computed with similar equations that use different points. Here are the equations to perform at control point (0, 0):

$$Q_{uu}(0,0) = 6\,(P_{00} - 2P_{10} + P_{20})$$

$$Q_{vv}(0,0) = 6\,(P_{00} - 2P_{01} + P_{02})$$

We need to do this computation in *x*, *y*, and *z* for each equation. This is a great place to take advantage of vector processing. We'll do this operation in parallel, computing both equations for *x*, *y*, and *z* simultaneously. First, we load both sets of control point positions into the vectors, as shown in pseudocode in Listing 2. Then just a few vector computations are performed and all coordinates are simultaneously calculated.

Note that we have the constants –2 and 6 stored in a vector constants (`vconst`) register, which makes it easy to multiply each slice in another vector by each scalar. Using vector processing we've reduced two additions and two multiplies for each of six coordinates to just two additions and two multiplies total.

We can perform this same process to compute Q_{uuvv}. But we'll have to pair up the operations. We have four control points, the corner points, which we need in order to calculate Q_{uuvv}. We can compute two separate control points simultaneously by jamming them into the same vector and doing vector operations. So we'll perform this process twice in order to compute Q_{uuvv} for all four points.

Surface Subdivision

For code simplicity, we're going to subdivide the surface iteratively, not recursively. We're going to subdivide many times, so let's make a function out of it. What do we need to pass to this function? Well, we'll need the data for the endpoints of the curve we're splitting, and a *du* value which is the distance in parametric space from the midpoint to the endpoint. This is 0.5 for our first subdivision. For simplicity, we'll pass in the value $(du)^2/2$ so we don't have to compute the square and multiply by one-half each time we use the function. We'll also stuff this value in a vector slice so that we can use it in vector computations. We'll call the vector which contains this value `vecdusqhalf`. Our function ends up looking like this (there will be one for u-curve splits, and one for v-curves):

```
void tesselSubdivide[U|V](Vertex v0, Vertex v1, Vector vecdusqhalf)
```

The microcode for the `tesselSubdivideU` function appears in Listing 3. The function `tesselSubdivideV` will be very similar.

The first thing you'll notice in this code is that we block together the Q_{uu} and Q_{uuvv} computations. We also block together the Q and Q_{vv} computations. That's because both of these are very similar computations. For Q_{uu} and Q_{uuvv} we're doing this:

$$Q_{uu}(u_{mid}) = \frac{Q_{uu}(u_0) + Q_{uu}(u_1)}{2}$$

$$Q_{uuvv}(u_{mid}) = \frac{Q_{uuvv}(u_0) + Q_{uuvv}(u_1)}{2}$$

The computations for Q and Q_{vv} depend on the prior computations, so we do them second. They look like this:

$$Q(u_{mid}) = \frac{Q(u_0) + Q(u_1)}{2} - \left(dusqhalf * Q_{uu}(u_{mid})\right)$$

$$Q_{vv}(u_{mid}) = \frac{Q_{vv}(u_0) + Q_{vv}(u_1)}{2} - \left(dusqhalf * Q_{uuvv}(u_{mid})\right)$$

Most of the opcodes you see in the microcode listing make intuitive sense. But one that bears explaining is `vmudm`. The RSP provides many multiplication operations. They vary depending on the sign of the operands and whether the operands are fractions or integers. The `vmudm` operation performs multiplication of signed integers by unsigned fractions. The resulting integer part of each vector slice computation is stored in the destination vector register (first operand), and the 32-bit integer/fraction results are stored in the accumulator slices.

This microcode currently is not optimized for dual processing, nor for accumulator storage of intermediate results. Vector loads and stores are scalar operations, so we could easily tighten this microcode up by executing loads and stores in parallel with vector operations.

Performance Figures

Calculating a regular grid of points on the Bézier surface using the standard double sum equation is a not a very efficient way to tessellate. Using floating-point arithmetic on the CPU, this process took 272,500 CPU cycles. While central differencing has a large performance cost at initialization, the subdivision step is very fast. Implemented on the CPU, it takes 70,400 CPU cycles to tessellate our surface.

When we moved this algorithm to the RSP, we made some sacrifices. We used 16-bit fixed-point arithmetic and took a hit for DMA-ing data to and from the RSP. But our algorithm, including RSP load and save time, runs in just 16,600 CPU cycles. And the CPU itself is free during this process to do other computations.

N64 Optimizations Keep the Platform Fresh

We've examined how Bézier surfaces can be implemented on the N64 using a central difference tessellation algorithm in microcode. The payback we got for choosing a more efficient surface tessellation algorithm and pushing it onto the RSP was substantial.

Technically, the N64 is still a powerhouse. Programming the microcode and taking advantage of vector processing gives developers the ability to implement algorithms that aren't feasible on much bigger and faster CPUs. In addition, learning to program the N64 now will give developers a big advantage when it comes to next-generation console development (including Nintendo's upcoming system, currently known as Dolphin), many of which use vector processing. If you're interested in becoming an N64 developer, or if you are an N64 developer and you want to know about microcode development kits, please contact Nintendo by sending e-mail to support@noa.com. ■

FOR FURTHER INFO

Books

Watt, Alan, and Watt, Mark. *Advanced Animation and Rendering Techniques: Theory and Practice*. New York: ACM Press, 1992.

Periodicals

Clark, J. H. "A Fast Scan-Line Algorithm for Rendering Parametric Surfaces." *Computer Graphics* Vol. 13 No. 2: pp. 289–299.

Game Developer

Sharp, Brian. "Implementing Curved Surface Geometry" (June 1999) and "Optimizing Curved Surface Geometry" (July 1999).

Gamasutra

For a detailed derivation and comparison of Bézier surface algorithms, see the expanded version of this article at http://www.gamasutra.com.

Bézier Surface Microcode Source

If you're a Nintendo 64 developer, log on to Nintendo's developer web site at https://www.warioworld.com.

LISTING 3. *Microcode for the* tesselSubdivideU *routine.*

```
##########################################################
# tesselSubdivideU
#
# Subdivide this curve in the U direction
#

# Surface Vertex structure offsets
.symbol   VERTEX_POS,                0
.symbol   VERTEX_UU,         8
.symbol   VERTEX_VV,         16
.symbol   VERTEX_UUVV,       24

# Register aliases
.name     pnew,              $10     # Position of output surface vertex (u)
.name     pminus,            $11     # Position of input left surface vertex (u-du)
.name     pplus,             $12     # Position of input right surface vertex (u+du)
.name     vecdusqhalf,       $v6     # Vector which contains 0.5*du*du in slice 0
.name     vecminus,          $v7     # Vector for storing left surface vertex info
.name     vecplus,           $v8     # Vector for storing right surface vertex info
.name     vecuus,            $v9     # Temp vector for UU and UUVV computation
.name     vecuushalf,        $v10    # Final results of UU and UUVV computation
.name     vecposvvsinter,    $v11    # Temp vector for POS and VV computation
.name     vecposvvshalf,     $v12    # Temp vector for POS and VV computation
.name     vecmulleduus,      $v13    # Temp vector for POS and VV computation
.name     vecposvvs,         $v14    # Final results of POS and VV computation

          .ent      tesselSubdivideU
tesselSubdivideU:

          # Do quu and quuvv computations together
          ldv       vecminus[0], VERTEX_UU(pminus)
          ldv       vecminus[8], VERTEX_UUVV(pminus)
          ldv       vecplus[0],  VERTEX_UU(pplus)
          ldv       vecplus[8],  VERTEX_UUVV(pplus)
          vadd      vecuus, vecminus, vecplus                    # Add endpoints
          vmudm     vecuushalf, vecuus, vecconst[1]              # Mul by one-half

          # Do qpos and qvv computations together
          ldv       vecminus[0], VERTEX_POS(pminus)
          ldv       vecminus[8], VERTEX_VV(pminus)
          ldv       vecplus[0],  VERTEX_POS(pplus)
          ldv       vecplus[8],  VERTEX_VV(pplus)
          vadd      vecposvvsinter, vecminus, vecplus            # Add endpoints
          vmudm     vecposvvshalf, vecposvvsinter, vecconst[1]   # Mul by one-half
          vmudm     vecmulleduus, vecuushalf, vecdusqhalf[0]     # uus/2 *(du^2)/2
          vsub      vecposvvs, vecposvvshalf, vecmulleduus       # Subtract...

          # Store everything
          sdv       vecuushalf[0], VERTEX_UU(pnew)
          sdv       vecuushalf[8], VERTEX_UUVV(pnew)
          sdv       vecposvvs[0],  VERTEX_POS(pnew)
          jr        return
          sdv       vecposvvs[8],  VERTEX_VV(pnew)
          .end      tesselSubdivideU
```

The Big S ee e: Re ce Ma ageme D i g Y C le P

by Michael Saladino

Developing games for consoles often requires tight resource management. Trying to fit all your textures, sounds, and polygons onto the machine is difficult enough, but then consider that your monsters, explosions, and bullets must live in this same place along with your program code. It's a tight fit.

But resource management is much more than just dealing with size constraints. You also have to properly allocate and free resources, which, if ignored, can quickly turn into hiding places for numerous bugs. Fortunately, there are ways to deal with these problems. It just requires planning and implementing some helpful tools.

First, let's see what we're getting ourselves into when we start talking about developing for a console. For this article, my focus is on the main memory, so we'll ignore video and sound memory.

The current champ of the console wars is the Sony Playstation, which means that if you're thinking about developing for consoles, you're probably considering this system. The Playstation's massive consumer market is the good news, but the bad news is that its market is the only big thing about it — the system itself is very limited when it comes to resources. The Playstation is a small system with only 2MB of primary memory. The Nintendo 64 isn't much better at 4MB. The newest kid on the block, the Sega Dreamcast, comes to the table with a more impressive 16MB of memory but it will sometimes have Windows CE taking a slice of that space. (Also, the recently-released specifications for the Sony Playstation 2 reveal that the system will have 32MB of main memory. But since it's currently in development, that might change.) In other words, the PC has a huge leg up on consoles as far as main memory goes and it doesn't look as if that will end anytime soon.

Keeping these restrictions in mind when developing a game is critically important, especially if you want to do simultaneous PC/console development (or worse yet, if you decide partially through your PC development to try porting it to a console). Many of the newest PC games require 24 or 32MB

Michael Saladino is currently the lead programmer on STAR TREK: HIDDEN EVIL at Presto Studios Inc. His previous work has found him at Volition Inc. during the development of DESCENT: FREESPACE and at Mobeus Designs Inc. Contact him at michaels@presto.com or just drop by his office for a late afternoon cocktail where he can be found exploring improved living through crushed velvet.

of loaded resources, so trying to fit that into a console is a daunting task. It's comparable to trying to fit an elephant into a Volkswagen. And don't expect consoles ever to catch up. PCs and consoles both use the same memory components and therefore on-board memory sizes for each are increasing at the same rate of approximately four times every three years. As long as PCs cost many times more than consoles, they will always have more memory.

So, you have a game that barely fits onto a PC, and then the producer starts talking about porting to a console. What do you do? Well, it's an old story in game development: A group of programmers just start building a game and before they know it, they're at 80MB of resources with no idea how they got there. Then they end up spending many man-months trying to shrink the game to fit on a consumer PC. At that stage, the idea to port the game to a console has to remain exactly that — an idea. You could never get it on a Playstation at that stage.

It's all too easy to allow this scenario to occur because most game programmers use systems with at least 128MB of memory, and many games can go well into their second half of development without anyone ever seriously looking at their memory footprint. Code, much like a gas, expands in volume to fill its container. Therefore it's up to the responsible programmer to set a clear goal for the size of the code base, know where it's going, and make sure that it stays on course. The basic ideas about memory management that I will explore are ones common to consoles and PCs. It's just more important to consider them while producing on a console game because the environment is so restrictive. Let's begin by looking at some useful coding techniques.

Memory Pools

Most data in a computer game consists of large numbers of common data types. Examples of this are the monsters in your world, bullets flying around them, polygons that make them up, the execute buffers that render them, and so on. You need to maintain lists of these data structures so they can be accessed quickly and efficiently. A good way to handle this data is with a memory pool manager. A memory pool manager is a piece of code that handles large collections of dynamically created data of a common type. With this layer of code, we can handle the allocation, de-allocation, and usage of these data elements. We can also keep track of important statistics about the data such as the greatest number of bullets that was ever in existence at one time during an execution of the game. Knowing these statistics can be very helpful in setting maximums that help compress your game's footprint onto a given system. I divide my memory pools into two separate versions. The first is responsible for handling data types which cannot have a maximum number of elements forced onto them. However, they must share the common trait that they are always allocated and de-allocated in two independent stages. The second type of memory pool handles data types that have a limited number in existence at any given time, which lends itself well to a temporal caching system.

Hungry common data types. These monsters' polygons are part of a cyclic memory pool.

Cyclic Memory Pools

Let's consider the first style of memory pool. To give you a clear idea of what type of data elements fall into the above description of this pool, it's any data list that is allocated element by element until it reaches its maximum growth for that cycle and is then completely destroyed so the process can begin again. Many data types in games actually fall into this category, and most involve frame-specific data because the frame is a convenient cycle. Examples of this data type include Direct3D execute buffers, polygons generated from 3D clipping, spans for scanline hidden surface removal, and AI transversal lists; all of which happen to be tied to the frame cycle. You build the data fresh every frame, and at the end of the frame you dispose of all that you created. Looking through a game's code base will most likely reveal many more algorithms that use this type of data.

One way to manage this data is to use a linked list that grows dynamically along with the data. This is easy but not very fast. First, you must call `malloc` (or `new`) for every element you create. For some data types this could easily be thousands of times per frame. But since these data elements need to be truly dynamic, a pre-allocated array will not work. Therefore, combining these two choices into a hybrid is what is called for. This is the first type of memory pool.

This memory pool allocates large numbers of elements with a single call to `malloc`. As new elements are needed, the memory pool system hands out pointers to memory blocks that have already been allocated as part of a previous chunk. If you run out, the memory pool will allocate another large block of elements for the next *n* requests for new elements. This keeps the number of system-level allocations to a minimum. The trick is maintaining the balance between saving time by creating more elements with each real allocation and losing memory with allocated space that goes unused. Each type of data element will probably work best with its own block size, which is determined by examining the data type's usage during an average game.

To reach this balance, you can customize the memory pool so that it grows and shrinks intelligently. For instance, if the data type tends to allocate the same number of elements every cycle, then there is no need to de-allocate the space every frame — just empty it and use it again. This results in even fewer system-level memory allocations. If the data type tends to spike up every few seconds, you can have the memory pool "prune" itself back to the average number of elements per cycle, thereby using far less memory.

Caching Memory Pools

The secondary form of memory pools manage data elements that are dynamic and cannot be allocated and de-allocated in two independent stages. Although this describes nearly every type of data list, there is one other restriction that this pool places on data types. The data must be able to handle an arbitrary maximum of elements to be enforced when a new element is requested. This memory pool is useful for "eye candy" — data elements such as bullets, explosions, and particle effects. Because console memory resources are very limited, wasting space on hundreds of bloody chunks is just not good. Therefore, a memory pool that keeps usage under control can be very helpful. Basically, this memory pool is a caching system. But many programmers only seem to use caches in reference to textures, when in fact they can be very useful with any number of data types.

Let's look at an example of what happens in my latest game, STAR TREK: HIDDEN EVIL. When an enemy drone is destroyed by a phaser blast, an explosion sprite is generated, a couple of smoke puffs are released, and a dozen spark particles are emitted. First, you don't want to allocate and delete every one of these elements each time a drone is destroyed. Take a quick look at some hyperactive console games and you can see that hundreds of these eye-candy objects are created every second. A good solution to this problem is to create a set number of each type of data element, say 100 debris particles, 30 smoke puffs, and so on, and then allow the memory pool to hand out new elements whenever they are needed without ever going over the set limit. This means that no system-level memory management occurs, which speeds up our code. We also know exactly how much memory each type of data takes up throughout the game.

What happens when we run out of elements of a given type? A simple age-based caching system can be used to destroy old instances in favor of new ones. While this could conceivably degrade image quality or even game play, it will usually go completely unnoticed if you carefully balance resource savings vs. game-play costs. An old bullet that has been flying for three seconds and still hasn't hit anything (such as one flying towards the sky) is probably not going to be missed if it disappears unnaturally. Smoke puffs and debris particles are also not going to be missed if they disappear a little too soon, especially since the player is most likely focusing on the area where new particles are being generated.

Memory Pools Integration

The integration of these pools into our code base should be simple enough since they are a very general class. For example, I wrote a cyclic memory pool type recently for handling dirty scanline updating in STAR TREK: HIDDEN EVIL. I had thousands of data elements, each of which represented a region on a scanline that needed to be refreshed. I created a memory pool of these data elements to request from when I needed them. Since I had multiple dirty scanline screens working simultaneously, I ended up saving tens of thousands of `new` and `delete` calls on peak frames. And by using very large chunks of a thousand data elements per pool, I had excellent cache locality coherency when walking the lists.

The integration of the caching memory pool might not be quite as obvious. All data elements that belong to these types of pools must derive from a common class. This common class contains one item: the field on which to base the caching algorithm. Earlier in the article, I implied that this caching scheme had to be based on the age of the object, but it can really be based on anything. Any heuristic that determines which element in the pool should be expunged in favor of a new element will do. Once you have all data elements deriving from the base class, the rest of the integration is the same as the other memory pool. Just stop allocating and deleting elements and instead request them from the appropriate memory pool system.

Once most objects in our game are based on the memory pool classes, we can expand their usefulness by turning them into statistic recorders to monitor memory use. Pools can keep track of how many of a given data element have been allocated, how many have been used, measure the peaks, and determine the averages. All this information can be gathered easily by the memory pool system and easily retrieved to help reduce your game's memory footprint. This data is also very useful for customizing the actions of each memory pool used by your game. For instance, you might find that you are requesting significantly more bullets per second than you first guessed and it's causing them to expire too early. Therefore, you raise your allowance for bullets.

An example of the real-world usefulness of these statistics can be seen in our memory reduction work on BENEATH. Our memory footprint was much too large at the end of the project, so we sent

Ensign Sovak is demonstrating caching memory pools by destroying an enemy drone. Sparks are great candidates for caching pools.

one of our programmers into the code to determine where we could cut. By breaking the memory usage down to the object level, we saw a tall spike at the data object "flare shards." Flare shards are pieces that fly off when a flare object explodes in the game. (A flare is shot from a flare gun, an object that many creatures use.) We discovered that the spike was caused by the fact that multiple creatures had flare guns, each flare gun had a pre-allocated clip of ten flares, and each flare had a pre-allocated clip of twelve flare shards. The result was thousands of flare shards just sitting in memory waiting to be used.

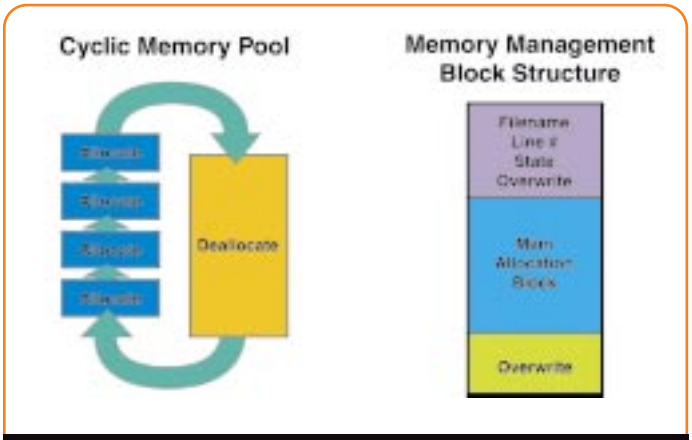


FIGURE 1. *Our memory manager's block structure.*

Our solution was to create a flare shard pool in which we allocated a set number of flare shards at the beginning of the game. Then, whenever a new flare shard was needed, it requested one from a memory pool. If the memory pool was full, the oldest flare shard was retired early and became the new flare shard. We ended up saving megabytes of data space. Something that small had actually gone unnoticed by us.

Memory Manager

As any experienced programmer can attest, memory errors can be some of the most difficult bugs to track down. And when you consider that consoles are often weak in terms of development tools, having your own code to help track down memory bugs can be quite a time saver. This is where a memory manager can be useful (see Figure 1). Normally, programs use the standard `malloc` and `free` (or their C++ equivalents, `new` and `delete`) when dealing with memory allocation; however, these functions do not help us track down memory leaks, prevent us from overwriting our memory bounds, or handle other common memory errors. So, let's look at a simple layer that we can wrap around `new` and `delete` in order to help us debug our code.

Let's examine how this layer will work. The meat of the code marks each memory allocation with a header and trailer containing very specific data that allows us to track the blocks. In the header we place a pointer to the source code file-name string that the block was allocated in and the line number it was allocated on. This lets us identify memory errors by generating useful debugging messages that lead the programmer straight to the problem. We also include a marker in both the header and trailer that allows us to track its current state and watch for boundary overwrites.

When an allocation occurs, our custom routine allocates memory using a `malloc` call with enough additional space to hold the header and trailer. The data portion of the block is cleared out to an uninitialized value. The header and trailer are initialized correctly with the file name, line number, and overwrite markers. This memory block is then put into a hash table to give us better searching performance when we need to find blocks in later stages, such as validation or freeing. The memory block is then returned with the appropriate adjustment for the header and the rest of the program is none the wiser that this has taken place behind the scenes.

Let's look at what happens when we free memory. First, check to see if the memory block has already been freed by looking at its header state. Also check the header and trailer markers to see if an unrecognized value has been placed there, implying that a boundary overwrite has occurred. This does not catch all memory overwrites because obviously the overwrite could have written our marker, and it would appear untouched. However, in all my years of using memory managers such as this, I've never seen that happen. If this memory block passes these initial tests, we make sure that the block is in our hash table. If we find it, then we have a legitimate `free` on a legitimate block. We free the data block by marking it as freed in its header state and then filling the block with a "freed value." The memory block is then removed from our hash table and we finally call `free()` to actually free the block on a system level.

Using these two functions, we can catch many memory errors when they are first introduced and not weeks later when we're trying to track down some very obscure, difficult-to-reproduce bug. We can also use these functions to track our total memory allocation, peak memory allocation, average memory allocation, and other useful code statistics.

Finding memory leaks is another big advantage to this code layer. At the end of your program, you can call the `FindMemoryLeaks()` function to find all memory leaks. The code to handle this is quite simple. By the time you call this function, a `free` should have been called for every `malloc` you made; therefore, any memory blocks that still exist in the hash table are memory leaks. Just walk the table, printing out every block along with its file name and line number information. You now have a printed list of every memory leak in your game.

With this layer written, how do we integrate this system into our code base? We could overload `new` and `delete`, or we could replace all our standard C memory calls with our own personal functions if we didn't want to use C++. The path I usually take is to use defines, such as this:

```
#ifdef _DEBUG
#define Allocate(x) \
  MemoryManager->Allocate(__FILE__,__LINE__,x)
#define Deallocate(x) \
  MemoryManager->Deallocate(x)
#else
#define Allocate(x) malloc(x)
#define Deallocate(x) free(x)
#endif
```

This method keeps you from having to type the file and line parameters for

every allocate function call. Also notice that the entire memory manager layer is skipped when not in a debug build, thereby speeding up the code base in the final release build. (And by that time, all the memory bugs should be gone, right?)

Jack confronts his arachnophobia to squash bugs. Of course, our memory manager can do that for him.

Resource Wrapping

The memory manager that we just looked at should save time when tracking down memory bugs. But what is the code really all about? It is just wrapper code; a custom high-level library that encloses a low-level library, such as the C memory manager, which provides improved functionality. By making sure that our wrapper is the only entry into the system, we can receive customized records that the low-level system was never designed to give.

Let's take this idea of the wrapper even further. There are many other resources that we can surround with wrappers. Why not wrap the file IO system? Or write a custom library instead of using `fopen()` and `fclose()`? This layer can be used to abstract packed files (many files merged into one for faster disk access) or keep endian switching straight between Macs, PCs, and consoles. You can also use it to track down unclosed files that can cause quite a bit of trouble, as a recent incident at Presto showed us.

Windows gives each application a maximum number of file handles, which when used up, can no longer be opened using standard C file access conventions. If you're opening files without closing them, you eventually use up these file handles and run out. A recent bug in our load/save system on STAR TREK ended up being a section of code that was not closing its files, and after five or six mission loads, we would run out of handles. We wasted a considerable amount of time on a bug that could have easily been avoided with a wrapper library reporting unclosed files. That's what wrapping systems are all about — adding more functionality to already-written systems.

Another benefit of writing wrapper code is that it helps us achieve our overall goal: porting between multiple platforms. By maintaining a custom application interface to a system, you avoid having to rewrite code throughout a project, and instead update only one library. For instance, if you had a game that used Windows-specific timer devices, you would have to track down all instances of such timer devices throughout your code base when porting to a console. Instead, write a wrapper class around the device that's used by the entire code base. Then, when it's time to port your game to another platform, you only have to touch one file by rewriting the timer wrapper. All other code in the game remains the same because it all uses your custom timer interface.

Specific Porting Issues

Let's take a moment to think about some specific issues when porting from a PC to a console. First, when developing for a console (especially a new console for which the development environments are not mature), you're often developing with DOS-prompt linkers and no support programs outside a basic C compiler. So when doing a port, take advantage of the PC's enormous library of development support before actually moving the code to the new platform. For instance, using NuMega's BoundsChecker is good place to start. For those who have never used it, BoundsChecker helps programmers find memory bugs similar to those discussed above, such as double `free`s, overwriting boundaries, and memory leaks. This program can help you remove memory errors before you even move the code base over to the console.

Another issue to think about is the difference between the Windows operating system and whatever operating system exists on the console you are porting to. (Or lack of operating system, as the case may be.) Windows is significantly more complex than anything you'll find on a console. (With the obvious exception of the Sega Dreamcast, which has Windows CE, but even that is a very stripped-down version of what you use every day on a PC.) As with all advanced operating systems, Windows can be quite forgiving of mistakes — sometimes too forgiving. Because if you make a mistake that Windows can survive, chances are your console won't. For instance, I've seen programmers not clean up after their memory allocation because they know that when they shut down their program, Windows will clean up their memory space. This is a horrible habit to get into, but from my experience, it seems common. Always work off the assumption that you do not have a safety net. When you type a `new`, match it with a `delete` immediately. When you open a file, make sure that you close it.

Other issues to consider, which I do not have the space to get into here, are problems associated with other resources, such as video space. Just as PCs will always have more main memory than consoles, they will also always have more video memory, because PCs owners are willing to spend more. The same problem exists for sound memory. Consoles don't yet have hard drives for high-speed data spooling or massive state saves, and the CD-ROMs they do have are usually considerably slower than what is common on PCs. In short, consoles are amazingly limited compared to PCs. But if you know what you're doing, you can make a game that's just as incredible as anything on a PC — sometimes even more so. ■

Code for the memory pool system and memory management wrapper can be found on the Game Developer *web site,* http://www.gdmag.com.

POSTMORTEM

I a i al Game
SYSTEM SHOCK 2

By Jonathan Chey

This is the story of a young and inexperienced company that was given the chance to develop the sequel to one of the top ten games of all time. The sequel was allotted roughly one year of development with its full team. To make up for the short development cycle and correspondingly small budget, the project was supposed to reuse technology. Not technology in the sense of a stand-alone engine from another game, but individual components that were spun off from yet another game, THIEF: THE DARK PROJECT. The THIEF technology was still under development and months away from completion when our team started working with it. To cap everything off, the project was a collaborative effort between two companies based on a contract that only loosely defined the responsibilities of each organization.

Jonathan Chey was the project manager and a programmer on SYSTEM SHOCK 2. He is also one of the co-founders of Irrational Games. Prior to founding Irrational Games, he worked at Looking Glass Studios and prior to that he received his Ph.D. in Cognitive Science from Boston University. He is currently working on his tan back in his hometown of Sydney, Australia. He can be reached at jon@irrational-games.com.

Add to these gloomy initial conditions the fact that the game from which our shared technology was derived slipped more than six months from the initially estimated date, that several developers quit during the project, that we didn't bring the full team up to strength until six months from the final ship date, and that we struggled with financial and business problems during the entire project. Having learned this, you might anticipate the worst. Strangely, SYSTEM SHOCK 2 shipped within two months of its targeted date and will, I hope, be recognized as a sequel worthy of its esteemed ancestor.

Let's step back and trace the origins of the companies and the project. Looking Glass Studios is familiar to many as the creator of a series of highly innovative titles including the original SYSTEM SHOCK, the ULTIMA UNDERWORLD series, the FLIGHT UNLIMITED line and TERRA NOVA, among others. Three years ago, Ken Levine, Rob Fermier and I were developers at Looking Glass, struggling with the aftermath of VOYAGER, an aborted *Star Trek: Voyager*–licensed project. At the time, Looking Glass was in financial and creative disarray after a series of titles that, though critically acclaimed, had failed to meet sales expectations, the latest being TERRA NOVA and BRITISH OPEN CHAMPIONSHIP GOLF. Frustration with the 18 months wasted on VOYAGER and a certain amount of hubris prompted three of us to strike out on our own to test our game design and management ideas. We wanted to nail down a rigorous and technologically feasible design, focus on game play, and force ourselves to make decisions rather than allow ourselves to stagnate in indecision. We wanted to run a project.

So we formed Irrational Games. After some misadventures with other development contracts, we unexpectedly found ourselves back at work with Looking Glass as a company rather than as employees. Initially, our brief was to prepare a prototype based on the still-in-development THIEF technology recast as a science-fiction game. The scope of the project was very wide, but we quickly decided to follow in the footsteps of the original SYSTEM SHOCK. Our initial design problem was how to construct such a game without the luxury of the actual SYSTEM SHOCK license, since no publisher had yet been signed.

Our initial prototype was developed by the three of us working with a series of contract artists. Our focus was on the core game-play elements: an object-rich world containing lots of interactive items, a story conveyed through recorded logs (not interaction with living NPCs), and game play realized through simple, reusable elements. This focus enticed Electronic Arts into signing on as our publisher early in 1998 — a fantastic break for us. It meant we could now utilize the real SYSTEM SHOCK name and characters.

Immediately, we went back to our original design, threw away some of the crazier ideas that had been percolating and began integrating more of the rich SYSTEM SHOCK universe into the title. That was the point at which the real development began.

The team at Irrational Games, from left to right: First row: Steve Kimura/Artist, Jonathan Chey/Project Director, Justin Waks/Multiplayer Programmer, Mauricio Tejerina/Artist, Rob "Xemu" Fermier/Lead Programmer, Dorian Hart/Designer, Lulu Lamer/QA Lead. Second row: Ian Vogel/Level Designer, Scott Blinn/Level Designer, Michael Swiderek/Artist, Rob Caminos/Motion Editor, Nate Wells/Artist. Third row: Mike Ryan/Level Designer, Ken Levine/Lead Designer, Mathias Boynton/Level Designer. Not shown: Gareth Hinds/Lead Artist.

It's the Engine, Stupid

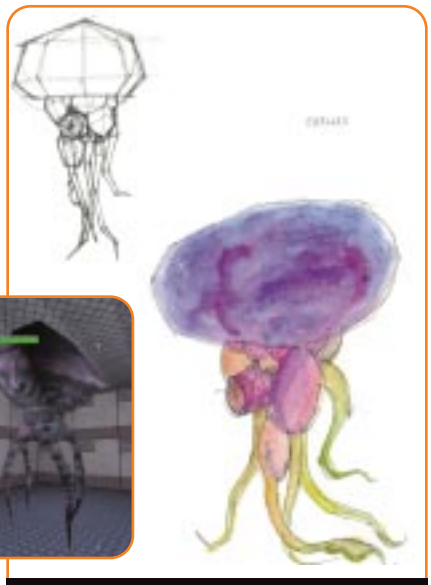
Nothing impacted the development of SYSTEM SHOCK as much as the existing technology we got from Looking Glass. This fact cannot be classified monolithically under the heading of "what went wrong" or "what went right," however, because it went both wrong and right. The technology we used was the so-called "Dark Engine," which was essentially technology developed as a result of Looking Glass's THIEF: THE DARK PROJECT (for more about its development,

SYSTEM SHOCK 2

Irrational Games LLC
Cambridge, Mass.
(617) 441-6333
http://www.irrational-games.com
Looking Glass Studios Inc.
Cambridge, Mass.
(617) 441-6333
http://www.lglass.com
Release date: August 1999
Intended platform: Windows 95/98
Project budget: $1.7 million
Project length: 18 months
Team size: 15 full-time developers, 10–15 part-time developers
Critical development hardware: Pentium II machines, 200MHz to 450MHz with 64MB to 128MB RAM, Nvidia Riva 128, Voodoo, Voodoo 2, TNT cards, Creative Labs' sound cards, Wacom tablets, Windows 95/98. Also used SGI Indigo workstations.
Critical development software: Microsoft Visual C++ 5.0, Opus Make, 3D Studio Max, Adobe Photoshop, Alias|Wavefront Power Animator, DeBabelizer Pro, RCS, Filemaker Pro, and Adaptive Optics motion capture software

see "Looking Glass's THIEF: THE DARK PROJECT," Postmortem, July 1999).

The THIEF technology was developed with an eye toward reuse, and I will refer to it in this article as an "engine." However, it is not an engine in the same sense as QUAKE's, UNREAL's, and LithTech. The Dark Engine was never delivered to the SYSTEM SHOCK team as a finished piece of code, nor were we ever presented with a final set of APIs that the engine was to implement. Instead, we worked with the same code base as the THIEF team for most of the project (excluding a brief window of time when we made a copy of the source code while the THIEF team prepared to ship the game). Remarkably, it is still possible to compile a hybrid executable out of this tree that can play both THIEF and SYSTEM SHOCK 2 based on a variable in a configuration file.



Concept Sketch of the Psi Reaver.

This intimate sharing of code both helped and hurt us. We had direct access to the ongoing bug-fixes and engine enhancements flowing from the THIEF team. It exposed us to bugs that the THIEF team introduced, but it also gave us the ability to fix bugs and add new features to the engine. Because we had this power, we were sometimes expected to fix engine problems ourselves rather than turning them over to Looking Glass programmers, which wasn't always to our benefit. At times we longed for a finished and frozen engine with an unalterable API that was rigidly defined and implemented — the perfect black box. But being able to tamper with the engine allowed us to change it to support SYSTEM SHOCK–specific features in ways that a general engine never could.

What Went Right

1. THE IRRATIONAL DEVELOPMENT MODEL. In our hubris after leaving Looking Glass, we formulated several informal approaches to development that we intended to test out on our projects. Most of these approaches proved to be successful and, I think, formed the basis of our ability to complete the project to our satisfaction.

First, we designed to our technology rather than building technology to fit our design. Under this model, we first analyzed our technological capabilities and then decided on a design that would work with it. This process is almost mandatory when reusing an engine. Sometimes it can be difficult to stick with this when a great design idea doesn't fit the technology, but we applied the principal pretty ruthlessly. And many of the times we did deviate we had problems.

Another feature of our development philosophy is that everyone participates in game design. Why? Because all three of the Irrational founders wanted to set the design direction of our products, programmers were able to resolve design issues without having to stick to a design spec, and we strongly emphasized game design skills when hiring all of our employees and contractors. In all our interviews, one of our most pressing questions to ourselves was "Does this person get games?" Failure to "get" them was a definite strike against any prospective employee. Ultimately, the team's passion for and understanding of games was a major contributor to the design of the final product.

The final goal of our development process was to make decisions and hit deadlines. We focused on moving forward, and we didn't allow ourselves to be bogged down. We desperately wanted to ship a game and believed that the discipline imposed by the rule of forward motion would ultimately pay off in terms of the final product quality as well as delivery date. While there are features in SYSTEM SHOCK 2 that could have been better if we had not rushed them (the character portraits for example), we still firmly believe that the game as a whole was made better by our resolve to finish it on time.

2. USE OF SIMPLE, REUSABLE GAME-PLAY ELEMENTS. The field of companies developing first-person shooters like id and Valve, among others, is impressive. From the outset we realized that we would have to work smarter, not harder, to make a game that could stand up in this market. It would be a futile attempt to create scarier monsters, bigger guns, or higher-polygon environments. Additionally, we realized that our design time and budget were very tight and that we would not have time to carefully hand-script complicated game-play sequences in the engine. Instead, in an attempt to shift the battlefield, we chose to focus on simple, reusable game-play elements. The success of HALF-LIFE, which launched while we were in the middle of SYSTEM SHOCK 2 confirmed our intuitions in this respect. We simply didn't have the time, resources or technology to develop the scripted cinematic sequences used by HALF-LIFE. We consoled ourselves with the knowledge that we were not even trying to do so.

This strategy melded very well with our acquisition of the SYSTEM SHOCK license, as the original SYSTEM SHOCK had already been down this road. We decided to expand on elements that we liked in SYSTEM SHOCK and then add similar new systems. Each such new system was evaluated rigorously in terms of game-play benefits, underlying technology, and design-time requirements.

For example, take the ship's security system. Early on we decided that we wanted to continue the surveillance theme from SYSTEM SHOCK, which we could leverage throughout the game to provide lots of game play for very little implementation cost. We realized that security cameras would be trivial to implement using existing AI systems (they are just AIs pruned of many of their normal abilities) and that once we had cameras that could spot and track the player, we would be able to build several game-play elements out of them. Cameras could summon monsters to the player, so much of the

game play consisted of avoiding detection by security cameras and destroying cameras before you were seen. Because cameras scan fixed arcs, the player can utilize timing to sneak by cameras, pop out and shoot them at the right moment, or get underneath them and bash them with a melee weapon. Once a player is spotted, monsters flood the area until the player is able to shut off the security system somehow or the system times out. This introduces the need to deactivate security systems via security computers that are scattered throughout the level.

This type of system was technologically simple to implement and required minimal design effort. While not completely formulaic, the basic procedure to set up a camera and security system could be shown to designers quickly using a few simple rules. From this one system and a couple of associated subsystems, we derived a large amount of game play without having designers create and implement complicated scripted sequences and story elements. When you throw together many such systems (as we did), you end up with a lot of game play.

3. Cooperative development. System Shock 2 was truly a cooperative development between Irrational and Looking Glass. Looking Glass provided the engine and a lot of infrastructure support (such as quality assurance), while Irrational handled the design, project leadership, and the responsibility for marshaling resources into the final product. Both entities contributed personnel to the development team. Inevitably, some friction arose from this process while we sorted out who was responsible for what. However, this cooperation was ultimately successful because both sides were interested in developing a great product, and we were able to compromise on most issues. (On the most mundane level, Irrational ended up providing late-night, weekend meals for its development team and for Looking Glass on some days during the week.)

Our cooperative arrangement was founded on a contractual agreement, but we avoided falling back on this contract in most cases. We preferred to resolve issues through informal discussions. Conceptually, Irrational was to be responsible for the development of the product and Looking Glass was to provide A/V content and quality assurance services.

Concept sketches of the Cyborg Midwife.

During the early stages of the project, a deal was worked out whereby a small number of Looking Glass personnel were subcontracted to Irrational when it was determined that Irrational's development budget could not cover all the System Shock 2 development costs, and as compensation for the late delivery of the Thief technology. Unfortunately, these personnel were not always available on time — a situation which caused us much concern. We knew that this "resource debt" could never really be paid off until Thief shipped — nothing is so difficult as prying resources away from a team that is trying to ship a product before Christmas. It wasn't until December 1998 that we first began to see some of these promised resources. However, these "resources" — real people — had just finished up Thief and were totally fried following the grueling crunch to ship Thief. The saving grace and reason that this arrangement was ultimately successful was that these developers were all talented and experienced and already knew the technology. Their addition to the team gave us a solid boost during the final months in our ship cycle.

The other benefit of the cooperative development agreement between Irrational and Looking Glass was that our respective engine programmers could share knowledge. The ability to walk over and quiz engine programmers about systems proved to be an invaluable benefit that more than compensated for the lack of a rigorously specified and documented engine. Without a formal understanding of the engine, we had to resolve engine issues in a personal and informal manner. This process relied on the personalities of the responsible individuals on the engine team. Thus, the Irrational programmers balanced their time not only according to the complexity of their tasks but also according to how much support was available from the engine side.

A psi-attack against a Hybrid in Engineering.

Overall, Irrational's relationship with Looking Glass was an unusually close one and ultimately successful as a result of our mutual respect and willingness to work with each other. Despite our partnership being based on a formal contractual arrangement, it was our ability to work flex-

ibly above this legal level that enabled the development to proceed smoothly.

4. Design lessons from System Shock. Though the System Shock license was wonderful, there were some problems. The biggest was simply the challenge of living up to the original. Fortunately, we had the freedom to pay homage to System Shock legitimately by reusing elements from it. Additionally, we had access to some of the original developers, including our own lead programmer Rob Fermier.

Cold comfort in Hydroponics.

As with most sequels, we faced the challenge of keeping the good elements of the original game while not blindly copying them. We knew that most players would want a new story set in the same world, with the same basic flavor as the original game, yet we also wanted to reach out to a broader audience. We resolved these issues by identifying the key elements that made System Shock so good and reinterpreting those elements using current technology. Some elements made it through largely unchanged (for example, the storytelling logs and e-mails, the über villain Shodan and her close involvement with the player throughout the game, and the complexity of the world). Other elements were reinterpreted (such as the look of the environment, player interface, and techniques for interacting with the world). A small number of items were simply cut, most notably the cyberspace sequences — we were fairly united in our opinion that these just didn't work well in the original.

Notably, as with the original System Shock, we opted to omit interactive NPCs in the game. System Shock eschewed living NPCs because the technology of the day was simply inadequate to support believable and enjoyable interactions with them. It's been four years, and that technology is still not available. So we continued the tradition of System Shock and provided players with background information using personal logs and e-mails gleaned from the bodies of dead NPCs.

Perhaps our biggest deviation from the original revolved around the player interface. It's commonly accepted that System Shock's interface, while elegant and powerful once understood, presented a significant barrier to entry. Our primary goal was to simplify this interface without dumbing it down. We devoted more design effort to this task than to any other system in the game, and it required many iterations before we were happy with it. We adopted a bi-modal interface in which there are two distinct modes (inventory management and combat/exploration) between which the player can toggle. This was a risky decision. This bi-modal model was mandated by our desire to keep the familiar and powerful mouse-look metaphor common to first-person shooters while retaining cursor-based inventory management. How we switched between modes became our biggest design challenge. Sometimes these mode changes are explicitly requested through a mode change key, and sometimes they are invoked automatically by attempting to pick up an object in the world. So far this system seems to be working well, though only time and user feedback will tell whether we really got it right.

5. Working with a young team. The System Shock team was frighteningly young and inexperienced, especially for such a high-profile title. Many of our team members were new to the industry or had only a few months' experience, including the majority of artists and all the level builders. Of the three principals, only Rob had previous experience in his role as lead programmer. Neither Ken, the lead designer, nor I, the project manager, had previously worked in these roles.

It's not totally clear how we pulled off our project with our limited experience. Partially, it must have been due to our ability to bond as a team and share knowledge in our communal work environment ("the pit"). To a certain extent, inexperience also bred enthusiasm and commitment that might not have been present with a more jaded set of developers. We also worked hard to transfer knowledge from the more experienced developers to the less seasoned individuals. Rob worked on an extremely comprehensive set of documentation for the functional object tools, as well as a set of exercises ("object school") to be worked on each week. These kinds of efforts paid back their investment many times over.

This is not to say that our progress was all sweetness and light. The art team, for example, floundered for a long time as we tried to integrate the junior artists and imbue a common art look in the team's psyche. We had a lot of very mediocre art midway through our project and the art team was stagnating. Ultimately, management had little to do with the art team's success — they were largely able to organize themselves and create a solid, original look.

On the management front, our inexperience was apparent. We blundered through the early stages of development with scheduling and management issues. A large problem was our failure to assign specific areas of responsibility and authority early on. Bad feelings arose as a result, which could have been avoided if we had clearly delineated areas of responsibility from the start.

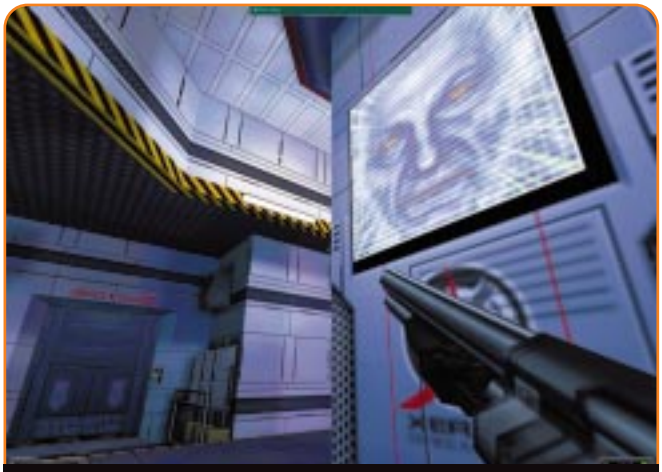
What Went Wrong

1. Poor level design process. Level design is a clearly defined professional activity in the game industry. It's a profession that mixes artistic and technical skills in equal measure, and the bar is raised on both fronts every year. Despite our understanding from

the very beginning that the level building would be a problematic part of the SYSTEM SHOCK development process, we didn't quite grasp how difficult and time consuming it would be, nor did we expect that it would eventually block the shipment of the game.

In hindsight, our failure to understand the amount of work needed to design levels is reprehensible given that we had seen the same problems emerge on THIEF, and that SYSTEM SHOCK 2 levels involved substantially more complex object placement than THIEF. I attribute this error mostly to our denial of the problem — we had a limited budget for level designers and there is a long training time required to get designers familiar with the complex Dark Editor. So we locked ourselves into working with the resources we had. Since each individual task required from the designer (apart from initial architectural work) was relatively simple, it was easy to believe that the sum total of work was also relatively small. What we overlooked was the fact that SYSTEM SHOCK 2 involves so many objects, scripts and parameters. As such, the work load on level designers was excessively large. In addition, we made a classic beginner's mistake and failed to provide adequate time for tuning in response to playtesting feedback. In SYSTEM SHOCK 2 this was particularly important because the ability of the player to reenter levels means that the difficulty of a level cannot be adjusted in isolation from the rest of the game. Often we had to impose global changes across all levels, which could be very expensive even when the change was relatively minor.

We took a novel approach to the level building process by attempting to apply design levels using a production-line method. Using this metaphor, we attempted to divorce the different stages of work on the level: rough architecture, decorative and functional objects, architectural polish, and lighting. It was not considered necessary for the same individual to be involved in all stages of this production process. This approach had positive and negative consequences. The advantages were that we could track progress on levels, we could "bootstrap" levels fairly quickly, and we could (in theory) swap individuals in and out of different tasks. The disadvantages are fairly obvious, and most stem from the fact that the various stages of level design are clearly not independent (for example, architecture is ideally built with an understanding of the functional objects that are to be used in the level). Although I think our process was necessary in order to get the game out on time, it probably detracted from the quality of some of the levels. In addition, psychological factors, such as lack of ownership and training issues (stemming from unfamiliarity with levels) speak very strongly against transferring people from one task or level to another. Nevertheless, there were several benefits of our procedure — mostly the ability to employ particularly talented individuals to pinch hit on particular levels, and the psychological benefits of completing architectural work early in the schedule.



Xerxes, central computer of the Von Braun.

Perhaps the rudest shock in our level building process came from our misunderstanding of what part of the process would prove to be most difficult. Architectural work was actually fairly simple, because we intentionally kept our spaces fairly clean and did not attempt anything too unusual. However, placing and implementing our objects was far more complex and involved than we expected. One difficulty that we encountered was educating our designers in what was expected from them in terms of game-play implementation. Most of our level builders had previously built QUAKE or UNREAL levels and were not familiar with the style of game play that we were trying to build in SYSTEM SHOCK 2. Partially this was because we were simply exploring a style of game play that we did not entirely understand ourselves. But it reflected a failure on our part to properly educate the designers. Building prototypical spaces, looking at past games and conducting more intensive discussions about game play will all be part of our future projects.

Our other major design hurdle was the instability and inscrutability of Dromed, the Dark Engine editor. Dromed is a cantankerous beast and many man-months were spent struggling with its idiosyncrasies. Perhaps our biggest problem stemmed from the lack of support in one crucial area — the part of the engine concerned with translating the designer-placed brushes into the basic world representation. Like many complex 3D engines, the Dark Engine suffers from troubling epsilon issues (data errors caused by rounding inaccuracies) and other glitches that crop up during level compilation. Because the programmer who implemented the basic renderer and world representation was not available during the majority of the SYSTEM SHOCK 2 development cycle, we had to work around these problems. It was often a frustrating process when the fundamental cause of the problems was not even known. Over time we developed a set of heuristics to avoid the majority of the glitches, but we were forced to lock down much of the level architecture before we wanted to in order to ensure stability.

2. MOTION CAPTURE DIFFICULTIES. The Dark Engine has a complex creature animation playback system and deformable mesh renderer. We encountered many problems with this piece of technology along our data integration path, and found quirks in the playback systems as well. Primarily, the system was hampered by the fact that data frequently had to be modified by hand, that mysterious bugs would appear in motions during playback

which had not been present in the source data, and that few tools were available for debugging and analysis. We were ill-equipped to deal with these kinds of problems, having devoted few resources to dealing with the technology problems.

Our primary animation source was motion capture data. We were nervous about the technology from the start and attempted to minimize our risk by concentrating primarily on humanoid creatures with a small number of interesting variants such as spiders and floating boss monsters. In retrospect, this was a very wise decision, as we had a lot of trouble even with this simple set of creatures. Motion capture technology and capture services were contracted from a local company, but unfortunately this company viewed its motion capture work primarily as a side business and did not display much interest in it. In fact, they cancelled this sector of their business during our project, and we had to fight hard to complete the sessions that we had already scheduled with them.

Our capture sessions were hampered by our inexperience with the technology and by the fact that we did not plan properly for the sessions. We hadn't defined key poses, rehearsed the motions, or ensured that our motions were compatible with the technology. Optical capture technology, the technology that we used, can be glitchy and has difficulty with motions that have obscured markers, as in the death motions that were necessary for SYSTEM SHOCK 2. Over the course of three sessions, we gradually refined our motions, but we spent a lot of time reshooting failed captures from earlier sessions.

Even in the best cases, most of our captures exhibit strange artifacts (feet pointing down through the ground, hands improperly aligned, and so on), whose causes are still unknown to us. In future projects we will hand-animate almost all of the data, and we will need to understand better what aspect of the conversion process introduced these artifacts into our final game animations, although the irregularities never appeared in our raw data. Motion capture technology, while highly efficient compared to hand animation, must be approached carefully to obtain good results.

3. IMPLEMENTING SCRIPTED SEQUENCES. Motivated by the dramatic scripted sequences in HALF-LIFE, we attempted to introduce similar elements into SYSTEM SHOCK 2. In doing so, we broke one of our rules: we tried to step outside the bounds of our technology. Although we attempted relatively simple sequences and ultimately got them working, they were time sinks, and the payback was relatively slight. For example, we scripted a hallucinatory sequence in which the player character rides through the interior of the alien boss-monster, known as the Many. This so-called "Many ride" was the source of innumerable bugs — the player would be thrown off the moving platform, manage to kill his projected self, bump into walls, and so on. We confirmed our intuition that the Dark Engine does not support complex scripted sequences well because the toolset (AI, moving terrain, and animation) is not optimized for this sort of behavior. The moral is, once again, to work with your technology, not against it.

4. INEXPERIENCE WITH MULTIPLAYER GAME DEVELOPMENT. Early in the project we were asked to identify the major risks associated with the project. Our number one candidate by far was the multiplayer component. This was the only new substantial engine feature that was to be added and it was a complicated piece of work. We were particularly nervous about this technology for a couple of reasons. First, it is usually much harder to make this kind of pervasive change to an existing piece of software than it is to build it in from scratch. Second, Looking Glass had no track record in shipping multiplayer technology and we were not confident that the development was fully understood.

Irrational did not want to introduce multiplayer support into SYSTEM SHOCK 2 because we considered it a tangential feature that did not contribute to our core strengths. However, marketing concerns dictated it, so ultimately we acquiesced. Our lack of enthusiasm for this feature contributed to its developmental problems because we failed to monitor its progress adequately or raise concerns when that progress fell behind schedule.

Because this was the first multiplayer product developed by Irrational or

Hacking the security system.

Looking Glass, we did not properly estimate the time required for the multiplayer testing. We did not devote adequate quality assurance resources to this feature. Too much time was spent testing the multiplayer features over the LAN and not enough over the more demanding modem connections.

Given the difficulties posed by the multiplayer technologies, the engine developers working on the task made great efforts, and their early results were promising. However, the early departure of one of the programmers, and the fact that he was not replaced, ultimately doomed any possibility of shipping the multiplayer technology with the initial SKU. Reluctantly, we opted for a patch that would be available at the same time as the single-player box reached shelves. Our cooperative multiplayer game will undoubtedly be fun and will probably be enjoyed immensely by a relatively small number of our customers. However, we wonder whether our failure to deliver a promised feature in the box will ultimately hurt us more than the absence of that feature from the start would have.

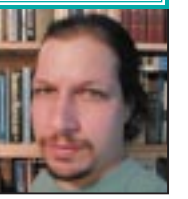
5. RUNNING A COMPANY WHILE BUILDING A GAME. As the principals of the company, Ken, Rob and I didn't really understand what it took to run a business and simultaneously work in that business. None of the Irrational founders started the company to be businessmen, and we have always believed that the ultimate health of the company depended on us all staying involved in the development process, which is, after all, what each of us enjoys and wants to do. Unfortunately, as anyone who has run a business knows, there is a lot more to starting and maintaining a company than sitting around at board meetings smoking cigars. From the mundane matters of making payroll, organizing taxes and expense reports to business negotiations and contract disputes, there is substantial overhead involved in running even a small company such as Irrational. In our naïveté, we did not factor these tasks into our schedules and the result was that they mostly became extra tasks that kept us in the office late at night and on weekends.

As a result of our misjudgment, we just had to work harder. Rather than enduring a crunch period of a few months, the entire last year of the project was our crunch time, as we struggled desperately to fulfill our jobs as programmers, designers, and managers as well as keep the money flowing in (and out) of the company. Our tasks were complicated further by the need to reincorporate the company from an S-corporation to an LLC during the final two months of the project (a legal maneuver designed to allow me, an Australian national, to be allocated company stock).

As well as destroying our personal lives, our failure to judge the magnitude of our task meant that we had to devote less time than we desired to every aspect of our work. My programming time was severely curtailed and I was able to spend far less time on SYSTEM SHOCK 2's AI than I wished. Simultaneously, I was unable to provide the level of direct management that I wanted, and I was forced to postpone company financial work until the end of the project or hurry it through. The results were less than optimal all around.

Ultimately, SYSTEM SHOCK 2 turned out better than I ever hoped it would. The final vindication for me was sitting in my office and playing the game in the final couple of weeks of the project, while waiting for EA to approve our final build. Despite the lack of sleep, the near-complete breakdown of my nervous system and the 18 months of time I spent working on the project, it was still fun to play. I like to think that we have managed to capture the feel of the original game by putting more game play into what initially looks like a fairly straightforward first-person shooter. It's been a great first project for Irrational Games and we look forward to doing even better the next time around. ■

by Greg Costikyan



A Platform for New Ideas: Why We Need an Indie Label Now

Last night, I went to see Kristin Hersh play at the Knitting Factory. You may not have heard of her, but the club was packed. Hersh releases her music on 4AD, an independent label.

It's not owned by Sony or Time Warner or BMG. Hersh doesn't get the kind of promo that major-label artists do — but she has more control over what gets released. Hersh is probably never going to chart — but she makes enough to live quite well, and reaches an audience of enthusiasts.

Tonight, I'm going down to the Angelika with my sweetie. It's New York's primary venue for independent films. The movies they show are never going to appear at your local Odeon or Sony theater; they'll be lucky to reach 500 screens in the States. But there are enough theaters like the Angelika to support a whole market for independent films — films that will never gross as much as a Hollywood blockbuster, but reach an audience of enthusiasts and earn enough to let many people live quite well.

At times, the music and movie industries look dull and played out and repetitive. You get the same damn formulas over and over, the same artists, the same directors. But that never lasts, and for one single reason: there's a venue for independent work. Indie labels and indie film companies experiment, at lower budgets, with the offbeat and original. And sometimes, they hit a nerve, build an audience, and ultimately rejuvenate the field. It happens continually in the music business, and it happened in film this past summer with *The Blair Witch Project*.

illustration by Jackie Urbanovic

Right now, the game industry looks dull and played out and repetitive. We get the same damn formulas over and over again. Yet another shooter. Yet another RTS game. Yet another racer.

A title as original or offbeat as SIMCITY or BALANCE OF POWER or M.U.L.E. — hell, or FROGGER — could not get funded today. Gaming needs a venue for independent work.

Last year, Miller Freeman did the industry a service by launching the Independent Games Festival. It's a place for "garage" developers to showcase product, get exposure, and maybe land a publisher. That's great, but it's not enough — because they're dealing with the same publishers as everyone else: EA, Interplay, GT, and others of their ilk. The majors will fund the tried and true, another shooter or RTS or racer. They'll happily exploit low-budget newbies who develop something that fits into slots they know how to sell — but they're not going to experiment with something new, something offbeat, something that will probably fail but just might rejuvenate the field.

The problem? There's no way to distribute an independent game. Yes, with a little effort, you can land a meeting with the buyer for CompUSA or Electronics Boutique or Software Etc., but they're not going to buy your game without the high-budget graphic glitz they expect from the majors, six figures in placement bucks, and a commitment to a major marketing campaign. The typical mall software outlet still has only 40 facings for computer games, and there are at least 1,500 entertainment titles published annually. Anything that doesn't fit the mold isn't going to get exposure.

Indie films work because there's a separate distribution channel parallel to the one for conventional film. Indie music works because there are a million record stores that cater to a million different tastes and a million small clubs where you can play to build an audience.

We've got nothing similar. We've

continued on page 63.

Greg Costikyan is a freelance game designer and writer. He has published 27 online, CD-ROM, board, and role-playing games, three novels, and a slew of short stories. He writes about games for a variety of web and print publications including Salon and Happy Puppy, and recently completed an analyst's report on the future of online games for Goodreports.com. Visit his web site at http://www.costik.com.

continued from page 64.

got to build it.

How? As recently as 1991, a typical computer game cost around $250,000 to develop. Graphics and sound have improved a lot since then, but computer games haven't gotten any better as games. You don't need $1.5 million in development funding to develop a first-rate game; you can do it on a $250,000 to $400,000 budget. You just can't get shelf space for a low-budget game.

But at that level, you don't need 100,000 unit sales. You can make money if you can get rid of 20,000 copies. And how tough can that be? Hell, 12 years ago, I sold more than 20,000 copies of a wonky little paper game called Paranoia through a wonky little distribution chain cobbled together out of specialty hobby game stores and comic shops. I doubt I had 500 points of sale in North America.

It can be done.

Some people are trying; Firaxis will be selling SID MEIER'S ANTIETAM! direct to consumers — no retail distribution. Michael Berlyn is bravely struggling to keep the text adventure alive through direct sales (see http://www.cascadepublishing.com).

But we need more. We need a company committed to publishing truly original, offbeat, cool product and building the channel for its distribution — instead of shoveling the same old crap to the same old stores.

Gaming needs an indie label — for the sake of its own health, to act as basic R&D for the entire field, and to find new gaming styles that can attract a large audience. Because development costs continue to spiral upward faster than unit sales and we have to find a way to break that iron cycle. But most importantly, because *I'm* tired of the same old same old and want to play something really cool and new. Don't you? ■