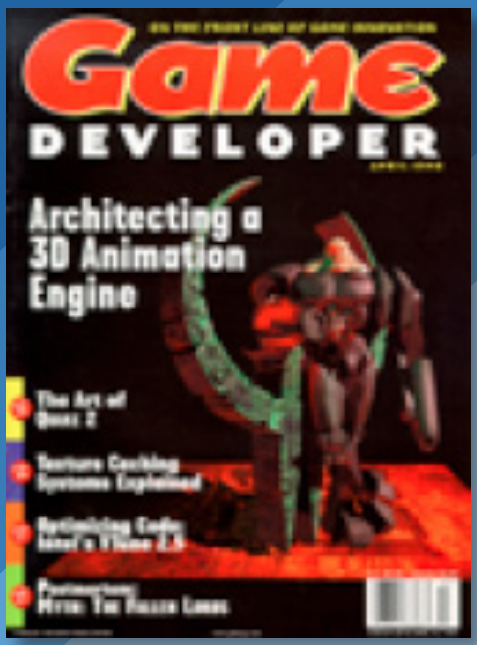
gd

GAME DEVELOPER MAGAZINE

APRIL 1998



UBM
Tech

Talent Shortage Means Many Uncertainties

Lately, there's been a lot of press coverage about the current shortage in information technology (IT) workers in the U.S. and its impact on the country's ability to keep up with the demand for high-tech goods and services. While none of the coverage I've read has addressed the impact on the game development industry specifically, there's no reason to think that our industry has somehow sidestepped the problem.

Sometimes it doesn't seem like we're facing a labor shortage of our own. The rec.games.programmer newsgroup is perpetually peppered with messages from people trying to enter the industry. In February, I wrote about a student-run game studio at UC Berkeley that's lined up dozens of members who are interested in getting into game development. Database programmers by day come home and work on their games at night (I know because I get several e-mails every day about their projects).

But let's consider the facts. The Information Technology Association of America (ITAA) and Virginia Polytechnic Institute released a study (available at www.itaa.org) in February stating that 10% of IT jobs (defined as programmers, systems analysts, and computer engineers) are now vacant. I don't know if they polled any companies such as Activision, Red Orb, or Interplay, but I don't think I'm sticking my neck out too far when I say that we're likely in the same boat.

Assuming that there's a shortage in our industry as well, I believe that the threat it represents to us is greater than, say, the neighborhood bank looking for a SQL developer. In addition, I think game development projects are more at risk of failure due to a talent shortage than other shrinkwrap products. Games are fundamentally different from other applications, and often they're extremely different from each other at the code level and the technology they use. In trying to create games that break genre molds and hit new performance heights, custom engines, optimized assembly routines, in-house utilities, and proprietary scripting languages have made jumping into a project midstream nightmarish for new hires.

The lack of programming talent affects animators, too. Many animators rely on programmers to create utilities, plug-ins, and other custom tools. Without enough programming talent to support animators, we might see the productivity of animators decline, the quality of their work degrade, or both. Fortunately, there seems to be a trend towards making customization easier for animators, with tools such as 3DS MAX R2 integrating scripting languages into their environments. Yet animators still face that learning curve, better though it may be than learning C.

For independent game development studios, the labor shortage problem is very serious. Imagine a programmer on a small team jumping ship halfway through development, and picture the kind of pressure that would put on the remaining staff. In this month's Postmortem, you'll read exactly how that scenario played out at Bungie during the development of MYTH. Bungie had to fall back on their company's network administrator to get some sections of the game completed.

A missed milestone and slipped ship dates can also cause serious financial repercussions, sometimes forcing management to put employee salary checks on their credit cards. While larger companies can spread their risk around by working on multiple projects, they face shareholder pressures or have to answer to parent companies if schedule dates are not met.

I don't want to sound like Chicken Little here — the industry won't collapse tomorrow because 10% of our programming jobs are vacant (assuming that number holds true for us). But it's worth bearing in mind that the problem exists, because as long as it does, your projects will face scheduling uncertainties. ■

ON THE FRONT LINE OF GAME INNOVATION
Game DEVELOPER

EDITOR IN CHIEF Alex Dunne
adunne@compuserve.com

MANAGING EDITOR Tor Berg
tdberg@sirius.com

EDITORIAL ASSISTANT Wesley Hall
whall@mfi.com

ART DIRECTOR Laura Pool
lpool@mfi.com

EDITOR-AT-LARGE Chris Hecker
checker@bix.com

CONTRIBUTING EDITORS Jeff Lander
jeffl@darwin3d.com
Josh White
josh@vectorg.com

ADVISORY BOARD Hal Barwood
Noah Falstein
Brian Hook
Susan Lee-Merrow
Mark Miller
Josh White

COVER IMAGE id Software

PUBLISHER Cynthia A. Blair
(415) 905-2210
cblair@mfi.com

MARKETING MANAGER Susan McDonald
AD. PRODUCTION COORDINATOR Dave Perrotti
DIRECTOR OF PRODUCTION Andrew A. Mickus
VICE PRESIDENT/CIRCULATION Jerry M. Okabe
GROUP CIRCULATION DIRECTOR Mike Poplardo
CIRCULATION MANAGER Stephanie Blake
CIRCULATION ASSISTANT Kausha Jackson-Craine
NEWSSTAND ANALYST Joyce Gorsuch
REPRINTS Stella Valdez
(916) 983-6971

un Miller Freeman
A United News & Media publication

CEO-MILLER FREEMAN GLOBAL Tony Tillin
CHAIRMAN-MILLER FREEMAN INC. Marshall W. Freeman
PRESIDENT/COO Donald A. Pazour
SENIOR VICE PRESIDENT/CFO Warren "Andy" Ambrose
SENIOR VICE PRESIDENTS H. Ted Bahr,
Darrell Denny,
David Nussbaum,
Galen A. Poss,
Wini D. Ragus,
Regina Starr Ridley
VICE PRESIDENT/PRODUCTION Andrew A. Mickus
VICE PRESIDENT/CIRCULATION Jerry M. Okabe
VICE PRESIDENT/SD SHOW GROUP KoAnn Vikören
SENIOR VICE PRESIDENT/SYSTEMS AND SOFTWARE DIVISION Regina Starr Ridley

www.gdmag.com

DirectSound3D Top Ten

I was disappointed by Rich Warwick's article in the January 1998 issue of *Game Developer* ("Avoiding a DirectSound3D Disaster," p.49). Suggestion #3, "Avoid using the `DuplicateSoundBuffer` call," misses the main reason `DuplicateSoundBuffer` exists: to conserve system resources when multiple instances of a sound must be played simultaneously. Rich is right to warn programmers that the call may fail unexpectedly, but he's wrong to categorize it as a mere "convenience" function.

Here are ten of my own suggestions, drawn from the sound engine for ROCKET JOCKEY, which was the first title to support DirectSound3D:

1. Approach the problem the same way you'd approach video rendering: create a list of potentially audible objects once per frame, sort it, and play only as many buffers as system performance can support.
2. Sort based on the needs of your game, but do consider distance, volume, and intrinsic priority. Sounds that are associated with visible events (gunshots) and sounds that are important cues (footsteps approaching from behind) are high priority. Sounds that exist simply to enrich the audio mix are low priority.
3. Don't play sounds that are redundant (three explosions at once); kill the one that's been playing the longest.
4. Stop playing sounds when their volume drops below a minimum.
5. When a sound is closer than a certain distance, disable 3D positioning, since the sound is "right on top of the listener."
6. Update the positions of sound objects and listener only once per frame. Use the `DS3D_DEFERRED` flag to tell DirectSound3D not to recompute the transfer functions until all the positions have been updated, then call `CommitDeferredSettings()` after culling the sounds that won't be audible this frame.
7. Define threshold values for pan, volume, and frequency, and don't update those parameters until the cumulative change exceeds the threshold value.
8. If your system can't support all the 3D buffers you need, you can crudely emulate 3D positioning using pan and volume. Pitch changes are expensive, so don't bother emulating Doppler shift. On a really slow system, you can fall back to just manipulating volume. ROCKET JOCKEY benchmarked system performance at installation time, and set a default Sound Realism value that the user could override.
9. Set the primary sound buffer format to match the format of your sound samples.
10. Avoid overhead from dynamic memory management (and memory fragmentation) by allocating a pool of sound buffers at the start of a level. Assign those buffers as needed.

Speak your mind! E-mail us at gdmag@mfi.com. Or write to *Game Developer*, 600 Harrison Street, San Francisco, CA 94107.

Dan Teven
Independent consultant

Responses to Hook's Column

Let me begin by applauding Brian Hook for taking the time to share how id's development process works from a programmer-management perspective. I'm sure that everyone, both technical and administrative, takes a great interest in how a company that created its own market develops products and how a world-class high-technology development team operates.

However, I do feel that the average reader should take it all with a fairly big grain of salt. What Hook describes is a classic example of "development by heroism," which is rampant in the game programmer culture, and not always to its benefit. This is referred to as level 1 of the SEI Process Maturity Model, the lowest possible level and where most teams stay. It demands that companies base their business around the particular core personalities; what results is a highly inflexible structure for improving or adding talent. At id there is no "Who's the man?" question because everyone there knows their place. But managing egos while still allowing growth for team member is a sticky problem not easily solved by seniority alone. While I'm not suggesting that the same structure used to create air-traffic controller software or Microsoft Word will create exciting and engaging games, the experience in this area shouldn't be completely discounted.

I do not question the validity of the id team model and I would never suggest that they change it. I do think that far too many business plans center around hiring or involving "the next John Carmack" and the belief that id's success can be recreated.

Chuck Walbourn, Vice President
Charybdis Enterprises Inc.

I just read Brian Hook's latest column in the February 1998 issue of *Game Developer* and I have to admit that I find it sad that he does not intend to continue writing for the magazine.

Especially because I've been pondering his final column quite a bit. I've been in this industry for twelve years now and I went through all the stages from programming in hexdumps on the old 8-bit machines to full object-oriented C++ implementations.

At first, when I read Hook's ramblings about C++, I was somewhat offended; I like C++ a lot. However, after about five minutes of thinking about his points, a cloud started to appear, and I thought "Hmm, there is some truth to that." The more I thought about it, the more truth I found in it, until eventually I thought "Darn, he's right. He is so right!"

While I believe that C++ has a valid position in the programming world, Hook's column made me think about my choices — even though it might not change them. And frankly speaking, before I start my next project, I will definitely sit down and seriously consider using C instead of C++ for the reasons he mentioned and many others besides — the language's speed and performance issues among them. I'm amazed how blind I've been, not even considering anything besides C++.

Guido Henkel, Senior Producer
Interplay Productions

INDUSTRY WATCH

by Alex Dunne

MICROSOFT HOSTED ANOTHER MELTDOWN in February, and the first evening Microsoft sponsored an event at the Seattle Gameworks. Ty Graham (Microsoft's "Mr. Meltdown") jokingly warned the crowd to be on their best behavior at that event, since Microsoft was sharing Gameworks with a group from Ralph Lauren Fragrances. To ensure that the Meltdowners did the company proud, Microsoft inserted automobile-style air fresheners emblazoned with the Meltdown logo in the attendee registration packets, and suggested that folks might want to wear them to really impress the RLF group.

CHRISTMAS UPS AND DOWNS. Electronic Arts had a merry Christmas. The company released an amazing 22 titles during its third fiscal quarter (which includes the holidays), helping to grow sales by 35% and profits by 50% over the same period last year. On the other hand, MicroProse announced that its third fiscal quarter ending December 31st wasn't as rosy. The company reported a loss of $10.9 million, on sales of $14.9 million. MicroProse acknowledged that its holiday titles missed their ship dates and didn't generate needed sales. The company expects to see red ink again this quarter, but also plans to release nine sequels to strong product lines, including FALCON and CIVILIZATION.

IMMERSION AND MICROSOFT have come to terms, and the result is that there will be better compatibility between Microsoft's Force Feedback Pro joystick and Immersion-technology-based sticks from companies such as CH Products and Logitech. Beginning with DirectX 6 and 7, the proprietary interfaces that Immersion and Microsoft currently have will be dropped in favor of one unified set of protocols.

HUZZAH! Simutronics and Renaissance Entertainment Corp. will collaborate in a

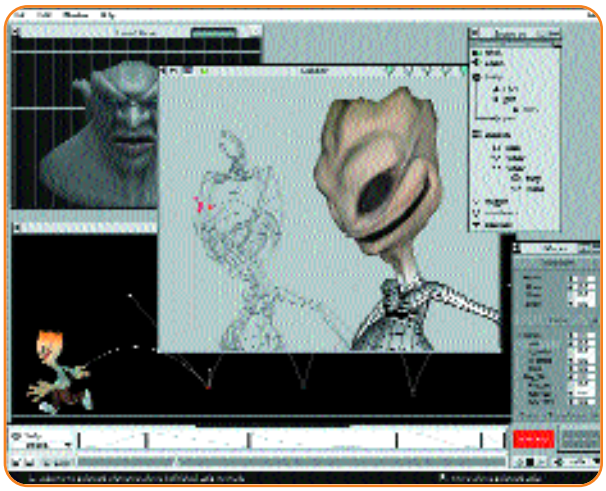
Project Dune

NICHIMEN GRAPHICS is planning the next major release of its 3D animation environment, N-World — currently known as Project Dune.

Dune strives to be holistic in nature and to adapt itself to each individual's working style (and with a minimum of palette popping and button clicking, due to a new object-oriented UI). Ease of use and simplicity are key elements of the program. It will allow you to view, edit, and work concurrently on different aspects of the scene data (UVs, geometry, and states, among others). Additionally, unlike other products that combine objects in linear time, Dune is a full animation system with nonlinear editing capabilities. As an artist, you will be able to embed motion into objects, but not explicity with respect to time or space. This grants you the ability to create re-usable content and direct it quickly. By separating all methods for creating content from one another (including animation and time operations) Dune allows you to edit any aspect of a scene without producing a ripple effect.

Project Dune will become simultaneously available for both Windows NT and SGI IRIX platforms. PC plug-ins also will run with the SGI installations. Due to the cross-platform user interface, artists and programmers will find no difference between the two versions of the product.

- Nichimen Graphics
  Los Angeles, Calif.
  (310) 577-0500
  www.nichimen.com



Game Control Interface

QUANTUM3D announced at the Amusement Trades Exhibition International (ATEI) show that it has developed a second-generation Game Control Interface (GCI) for PC-based coin-op, location-based entertainment, and visual simulation applications.

This new generation of GCI is designed to interface with coin-op industrial input/output devices and personal computers that run Windows 95/NT and DOS. The GCI bridges the gap between traditional arcade-style controls — such as analog joysticks, steering wheels, buttons, and coin mechanisms — and Intel architecture PCs. It allows both arcade and consumer game developers to design video games for "out-of-home" venues without worrying about the specific electrical, mechanical, and software aspects of coin-op I/O devices. For developers, Quantum3D has developed a GCI Developers Kit (GDK) that provides a comprehensive development environment that includes everything required to create a new PC-based application with coin-op I/O, or to add coin-op controls to an existing aplication.

ASTS

OF GAME DEVELOPMENT

Designed to support development under MS-DOS and Windows 95/NT, the GDK includes a GCI with internal PC mounting cable assemblies and brackets, the GCI to JAMMA adapter, Direct Input and GCI driver software binaries, test programs, wiring harness design guide, and a Quantum3D Gameframe. The suggested retail price for the standalone GCI is $250. The pricing for the GCI Developers Kit is $2,795.

- Quantum3D Inc.
  Santa Clara, Calif.
  (408) 919-9999
  www.quantum3d.com

Lucidity RT

DIGITAL MEDIA INTERACTIVE recently released Lucidity RT for interactive video authoring.

Designed as an advanced plug-in for Kinetix's 3D Studio MAX, Lucidity allows you to create fully interactive visualizations and multimedia titles with broadcast-quality graphics and photo-realistic detail. First, you use MAX to develop a navigable space or objects to be visualized. You can then use Lucidity RT to create a "visualization model" made up of "motion kernels" or video segments. Lucidity directs the rendering of all required views, and then encodes the rendered views in MPEG to create the motion kernels, placing these in an interactive video database for access during playback. Individual motion kernels are readily identified for editing, re-encoding, or post processing. Once you've created these interactive environments, you can port them to, and play them on, your stardard notebook, laptop, and desktop PCs.

Lucidity RT runs on Windows NT and has a suggested retail price of $1,295.

- Digital Media Interactive Inc.
  San Mateo, Calif.
  (650) 655-4824
  www.dmix.com

Lucidity RT allows you to create photo-real interactive environments.

TrueMotion 2.0

THE DUCK CORPORATION announced the release of its TrueMotion 2.0 Compression Tools at Microsoft's Winter Meltdown 98 in February.

DirectX Media 5.1 offers APIs and run-time services for game developement, multimedia applications, development tools, and Web content. The TrueMotion 2.0 playback filter, included in DirectX 5.1, provides game and multimedia developers with a way to incorporate full-motion video and interactivity into their products.

TrueMotion is a set of video compression and decompression components available for DirectX for Windows 95/NT. In addition, the Truemotion 2.0 playback filter is scheduled to ship as part of Windows 98. Duck's TrueMotion and TruePlay technologies are also avialable for Macintosh, and Sega Saturn, among other platforms.

- The Duck Corporation
  New York, N.Y.
  (212) 692-2000
  www.duck.com

new advertising campaign. REC is the organizer of various Renaissance faires throughout the U.S., events which encourage attendees to engage in live role playing. In their agreement, Simutronics' role-playing games will be featured on REC's web site and in all of their advertising for the upcoming faire season.

IT'S TOUGH TIMES OVER AT S3. **The company announced that it's laying off approximately 15% of its employees (100 people) to slash operating expenses. The company has said that it isn't anticipating a profitable first half of 1998.**

3DLABS FILED A LAWSUIT charging Texas Instruments with breach of contract and misappropriation of trade secrets belonging to 3Dlabs. 3Dlabs claims that, in breach of the parties' written license agreements and in violation of the trade secret provisions, TI posted confidential and proprietary information belonging to 3Dlabs on the Web, and distributed thousands of printed copies of confidential and proprietary information belonging to 3Dlabs.

STATS O' THE MONTH DEPT.:

- **Research firm Frost & Sullivan reported that industry revenues for online gaming grew to approximately $93.3 million in 1997,and projected that revenues for 1998 will be $169.6 million – an increase of over 80%.**
- **Worldwide sales of hardware and software for the home interactive entertainment industry surpassed $23 billion in 1997, with the United States representing nearly 40% of the total market, according to the research firm Access Media International.**
- **Mercury Research predicts that the number of 3D-enabled accelerators shipped in 1998 will more than double to 75.3 million.**

ION STORM is licensing yet another highly-touted game engine – the UNREAL engine from Epic MegaGames. This engine will be used in Warren Spector's upcoming 3D espionage game, code-named Shooter, which will be published late this year by Eidos. ION Storm is in negotiations with Epic to use the engine for future games. It has already licensed the QUAKE 2 engine for its upcoming DAIKATANA and ANACHRONOX titles, and it's using AnimaTek's voxel technology in DOPPELGANGER.

Slashing Through Real-Time Character Animation

Last time we left off, you were hanging out on a bridge in quaternion space. Quaternion space is like the space between the handout desk and the deal room at any booth of any publisher at E3. It's a concept you can grasp, but it's really tough to visualize being there.

As you'll recall, I finished up with an OpenGL application that converts Euler angle data to quaternions and then displays the results. Now I want to extend this application to allow for the interpolation of two keyframed positions. It struck me that I should create a custom 3D first-person interface that effectively demonstrates interpolation.

The Task

As the project technical lead, I am asked to create an interface for a first-person fighting game. However, the design calls for allowing the player to create custom attacks in some sort of pregame editor. The player does this by manipulating an arm consisting of an upper arm, a lower arm, and a hand with a weapon. The player positions this arm into two poses. One pose is the beginning of the attack move and the other pose is the end of the attack move. During the game, this custom action is triggered and creates a smooth attack. The player's effective use of this interface determines the effectiveness of the move. Several of these moves are then combined to create a unique fighting experience.

Alright, so it's not revolutionary, but it's a well-defined task with a pretty clear path of attack. As technical lead, I like that. So how do I get started?

When not bending the bones of some strange alien creature, Jeff can be found hanging out at his studio at the beach. See if you can smack some sense into him by writing to jeffl@darwin3d.com.

Clearly, the problem revolves around the interpolation of the arm positions.

Interpolation

As I discussed last month, one of the key benefits of using a quaternion representation is the ability to interpolate between keyframes. Nick Bobick, in his article in the February 1998 issue of *Game Developer*, discussed interpolation of quaternions ("Rotating Objects Using Quaternions," pp.38-39). Bobick described the use of Spherical Linear Interpolation (SLERP) to achieve smooth interpolation. For very small interpolation, he mentioned that it's a good idea to use simple Linear Interpolation (LERP). Being the hardcore game programmers that you are, you may ask, "Why not use LERPs all the time and avoid the expensive math that SLERPs require?" The reason is that unit length quaternions describe a 4D hypersphere. If I were simply to interpolate between the two keyframes in a straight line, I would be cutting across the arc of that sphere (Figure 1a). As you can see, in-betweens that are evenly spaced on the hypersphere create nonlinear positions on the LERP-line. Alternately, the effect of evenly spacing out in-betweens along the LERP-line would create an animation that would appear to move faster as it traveled across the middle of the interpolation (Figure 1b). This may not always be a bad thing, so it's quite easy to adjust between LERP and SLERP.

The code in Listing 1 gives me the basis for creating my 3D interface. By applying these routines to a three-bone hierarchy, I get a smooth attack from any two keyframed positions. To implement this in OpenGL, I only needed to modify my display routine a little bit. The critical section is in Listing 2, and you can see the results in Figures 2a-c.

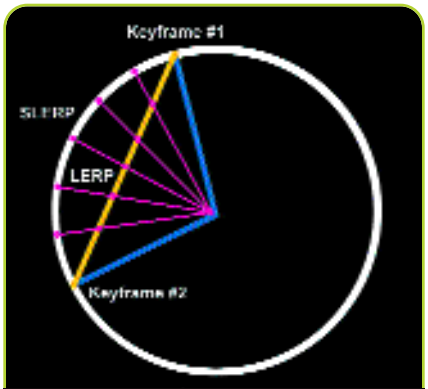


FIGURE 1a. *Spherical Linear Interpolation (SLERP) between quaternions.*

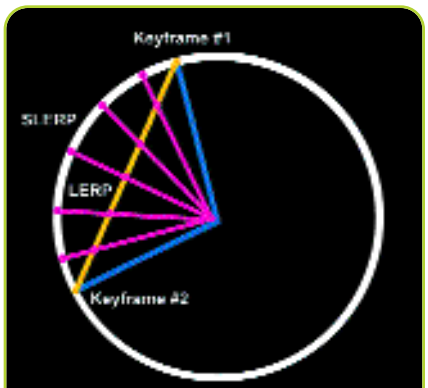


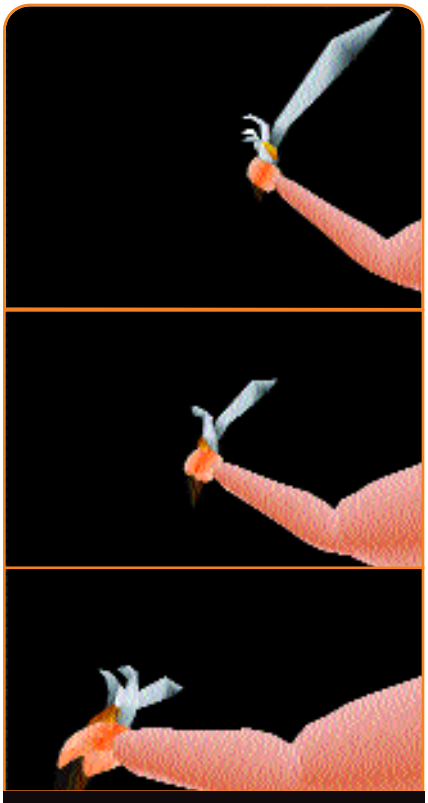
FIGURE 1b. *Linear Interpolation (LERP) between quaternions.*

Let's Recap

Let me take a moment to recap what I've covered in my past two columns. I started by taking motion capture data and applying it to a skeletal system. I then created a method for converting Euler angles to quaternions. Then, by using quaternion interpolation, I created smooth in-betweens for the skeletal system. Lastly, to make things look a bit more interesting, I attached 3D objects to individual bones.

One remaining problem is that the arm is created from three separate objects — and it shows. The points at which the different objects are connected look a little rough. This problem has been plaguing real-time 3D graphics for some time now. In fact, many successful games live with this and get quite good results (VIRTUA FIGHTER, TOMB RAIDER, and JEDI KNIGHT come to mind). However, to combat this problem, many artists design their characters to disguise the fact that the bones are composed of separate objects. This is done through clever texturing or modeling, and explains why so many real-time 3D characters wear armor or tank top shirts.

If the character was created from one single mesh, we wouldn't have any of the problems that separate objects create. In a 3D graphics package such as Softimage, Alias, or 3D Studio MAX, I could create a mesh and animate it with the software's skeletal



FIGURES 2a-c. *Keyframe #1, 50 percent interpolation, and keyframe #2.*

LISTING 1. *Interpolation of Two Quaternions.*

```
////////////////////////////////////////////////////////////////////////////
// Function:        SlerpQuat
// Purpose:         Spherical Linear Interpolation Between two Quaternions
// Arguments:       Two Quaternions, blend factor, result quaternion
// Source:          Watt and Watt, Advanced Animation, p. 364
//                  NOTE: This fixes a bug in their code
////////////////////////////////////////////////////////////////////////////
void SlerpQuat(tQuaternion *quat1,tQuaternion *quat2,float slerp, tQuaternion *result)
{
/// Local Variables ///////////////////////////////////////////////////////
        tQuaternion quat1b;
        double omega,cosom,sinom,scale0,scale1;
////////////////////////////////////////////////////////////////////////////
        // USE THE DOT PRODUCT TO GET THE COSINE OF THE ANGLE BETWEEN THE QUATERNIONS
        cosom = quat1->x * quat2->x +
                        quat1->y * quat2->y +
                        quat1->z * quat2->z +
                        quat1->w * quat2->w;

        // MAKE SURE WE ARE TRAVELING ALONG THE SHORTER PATH
        if ((1.0 + cosom) > DELTA)
        {
                // IF THE ANGLE IS NOT TOO SMALL, USE A SLERP
                if ((1.0 - cosom) > DELTA) {
                        omega = acos(cosom);
                        sinom = sin(omega);
                        scale0 = sin((1.0 - slerp) * omega) / sinom;
                        scale1 = sin(slerp * omega) / sinom;
                } else {
                        // FOR SMALL ANGLES, USE A LERP
                        scale0 = 1.0 - slerp;
                        scale1 = slerp;
                }
                result->x = scale0 * quat1->x + scale1 * quat2->x;
                result->y = scale0 * quat1->y + scale1 * quat2->y;
                result->z = scale0 * quat1->z + scale1 * quat2->z;
                result->w = scale0 * quat1->w + scale1 * quat2->w;
        } else {
                // SINCE WE FOUND THE LONG WAY AROUND, USE THE SHORTER ROUTE
                result->x = -quat2->y;
                result->y = quat2->x;
                result->z = -quat2->w;
                result->w = quat2->z;
                scale0 = sin((1.0 - slerp) * (float)HALF_PI);
                scale1 = sin(slerp * (float)HALF_PI);
                // MULT BY THE SCALE
                result->x = scale0 * quat1->x + scale1 * result->x;
                result->y = scale0 * quat1->y + scale1 * result->y;
                result->z = scale0 * quat1->z + scale1 * result->z;
                result->w = scale0 * quat1->w + scale1 * result->w;
        }
}
// SlerpQuat ////////////////////////////////////////////////////////////////
```

LISTING 2. *Applying quaternion rotation in OpenGL.*

```
// CONVERT THE TWO KEYFRAME ROTATIONS TO QUATERNIONS
EulerToQuaternion(&curBone->p_rot,&primaryQuat);
EulerToQuaternion(&curBone->s_rot,&secondaryQuat);
// INTERPOLATE BETWEEN THEM BY A BLEND FACTOR 0.0 - 1.0
SlerpQuat(&primaryQuat,&secondaryQuat,m_AnimBlend,&curBone->quat);
// QUATERNION HAS TO BE CONVERTED TO AN AXIS/ANGLE REPRESENTATION
QuatToAxisAngle(&curBone->quat,&axisAngle);
// DO THE ROTATION
glRotatef(axisAngle.w, axisAngle.x, axisAngle.y, axisAngle.z);
```

deformation system. This would create very seamless characters that could be animated very quickly by a game engine. In fact, this method has been used by QUAKE (and its genetic offspring) quite successfully.

However, by predeforming the characters to animate them, I lose the key benefit of real-time 3D — flexibility. By sticking to a hierarchy of bones, I'm able to apply unique motion capture data, interpolate between keyframes, and incorporate many things that I haven't talked about, such as real-time inverse kinematics, dynamics, and motion blending. So how do I achieve the key benefits of a skeleton without having the ugly seams that come with it?

Skin Them Bones

The answer is to stretch a single skin over the bones in the skeleton. While this is a fairly advanced feature in most 3D modeling packages, the technique behind it is really quite easy. As a good starting point, let's consider associating each vertex in the skin mesh with an individual bone. The influence of that bone directly effects its associated vertex . Thus, when you rotate a bone, it rotates the associated vertices about the root position of that bone. You can see the effect of this process in Figure 3. In this image, two bones define the hierarchy, and each bone has eight vertices associated with it. While this technique creates a seamless, deformable mesh with very little processor overhead, it has one drawback. At the point where the two bones meet, the skin is stretched a bit. While this may be fine for many applications, with extreme motion, this stretch is very unrealistic.

The solution to this problem is to add a few more vertices to the model and "weight" the individual vertices. This means that for each vertex in the model, you assign a certain percentage of its influence to each bone. While many vertices may be assigned 100 percent to an individual bone, some may be assigned 50/50 between two bones. By blending the influence of different bones, you can achieve a very smooth skin. In some extreme cases, you may even need to weight a vertex between three or more bones, but in general, two is sufficient.

Figure 4 shows how a fully weighted mesh could be applied to the same two-bone system. Because of the calculations needed to handle the weighting, this system is a bit more processor intensive then basic skinning. However, for a main character or opponent, the smooth results and flexibility are worth the increased processor load.

The remaining question is, How do I implement a fully weighted mesh applied to a hierarchical skeleton in an immediate mode API such as OpenGL? Well, that's going to be the topic for next time. However, some of you may be itching to get started. Since I have now covered all of the main pieces, for your homework, see if you can figure out an efficient way to calculate those vertex positions, and I will work it out next month. Try to wrap your brain around the underlying concept, and then we'll work out the details together next month.

The sample application for this month (on the *Game Developer* website at www.gdmag.com) allows you to keyframe two positions for a three-bone arm and interpolate smoothly between them. ■

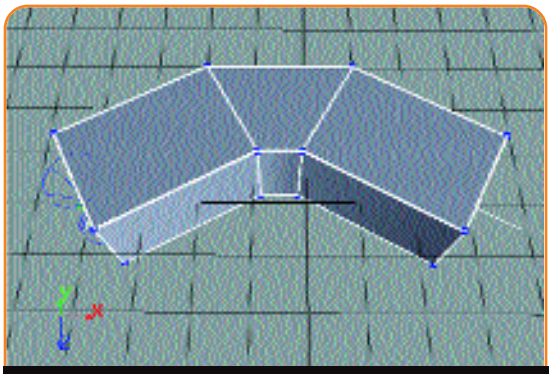


FIGURE 3. *Two bones weighted 100 percent to each vertex.*

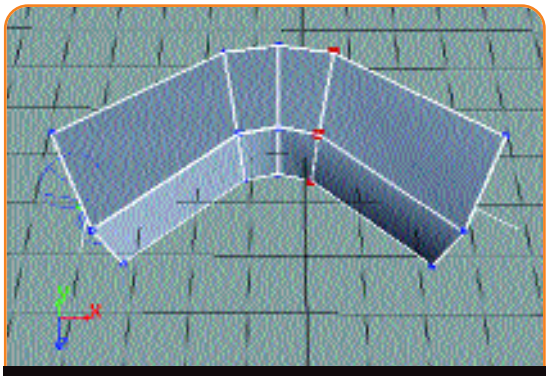


FIGURE 4. *Two bones weighted to each vertex 100-66/33-50/50-33/66-100.*

REFERENCES

For my quaternion SLERP code, I have used *Advanced Animation and Rendering Techniques*,(ACM Press, 1992) by Watt and Watt as a starting point. This is a very good book that covers many topics not touched by any other text. However, as I've tried different concepts in the book, I have found a few errors. The code sample on interpolating quaternions is a case in point. When negating an input quaternion to find the shortest arc, they negate the wrong one. The source that accompanies this article corrects that error. Problems such as these are a good reason to study the underlying concepts and follow sources for any new technique. This can save you a great deal of frustration when things do not work out as they should.

Art Direction: A Touching Story

Art direction is the communication between an original designer and the executor of the design. It's not the design process, and it's not the final design's execution. This month, we'll peek at the perils and delights of game art direction.

For example, let's say you want a portrait of your pet lizard. Your idea is to put the lizard on a nice branch with a desert scene as a backdrop (that's the design). You hire a professional photographer to actually take the photograph. You spend 15 minutes explaining how you want the shot to look and drawing a simple layout sketch (that's art direction). The photographer sets up the shot, takes some prints, and shows them to you (that's art execution). You provide criticism and ask the photographer to retake the shot (that's art direction).

Now that we have an idea of the scope of art direction, let's look at how it works in the game industry (obviously, this is a fictional story).

Kotz and Amy's Little Issue

Kotz is a wound-up monkey of an art director with a couple of solid hits under his belt. His enthusiasm is legendary — with spittle flying and arms flapping, he spews out quirky artistic ideas nonstop and then leaps onto his drafting stool and slashes out his trademark designs, hooting "Ooo-ooo! " when he really gets in the groove.

Kotz was hired to design artwork for a game project called "DreamFly," a cartoonish, dreamy flight simulator that feels like a kid's book. There are little fairy creatures fluttering around a big, puffy airplane. The plane zooms over surreal blobs of landscape — you know, concept stuff. Kotz loves this idea, and has already scrawled out a huge pile of sketches for the game artwork.

Now the project is ready to start, and Kotz's job is to find artists to build his design. He hires Amy, a skeptical veteran game artist, to create 2D sprites of the fairies dancing around the airplane. Kotz's initial direction to Amy consists of handing her a crude and childishly executed sketch of a fairy.

"So hey! Check it out! I bet you're just *itching* to build this little bugger!" Kotz chirps as Amy stares at the sketch in disbelief.

Let's freeze that frame and take a look at the problems so far. The sketch is completely inadequate for character definition — there's not enough information. But there's another important problem developing here.

Problem 1: Staying Friends

Amy and Kotz are forging a relationship. This is an important interaction, and it's not going well. Neither of them is taking care to create a solid foundation for their working relationship, even though that's what they both want.

As Amy's boss, Kotz wants to communicate enthusiasm and good vibes, but he has set himself up for disaster by approaching the problem too lightly. Since this is the first project he and Amy will be collaborating on, he should have assessed the situation more carefully. He should get Amy's input on the art-direction process and make her feel comfortable giving feedback. Right now, Amy can't criticize anything without implying that Kotz screwed up — but she needs to say *something*. That's a lose-lose situation.

What are Amy's options, given this situation? One option would be a response such as, "Umm... Kotz, I can't draw this." Her low, measured tones totally puncture his mood. "This sketch is, uh, really rough." Kotz's heart sinks. "What's wrong? It's a cool little fairy, right?" Amy's exasperation wells up as she dramatically sighs, "Oh, good grief... Kotz, this drawing is so far from a complete design! I can't work miracles."

Amy's reaction is justifiable since Kotz has obviously made some mistakes, but that kind of self-centered attitude won't get her far in her career. Even if she hates Kotz and is planning to quit, she'll do better if she doesn't burn bridges. How can she be honest without burning bridges? Amy can prevent conflict by buffering her negative reaction.

David Packard (of Hewlett-Packard fame) wrote a book called *The HP Way* (HarperBusiness, 1995), which describes Bill Hewlett's three-step process for critiquing ideas without dampening enthusiasm:

> Upon first being approached by a creative inventor with unbridled enthusiasm for a new idea, Bill immediately put on a hat called

Josh White runs Vector Graphics, a real-time 3D art production company. He wrote Designing 3D Graphics *(Wiley Computer Publishing, 1996), he has spoken at the CGDC and cofounded the CGA, an open association of computer game artists. You can reach him at josh@vectorg.com.*

"enthusiasm." He would listen, express excitement where appropriate and appreciation in general, while asking a few rather gentle and not too pointed questions. A few days later, he would get back to the inventor wearing a hat called "inquisition." This was the time for very pointed questions, a thorough probing of the idea, lots of give-and-take. Without a final decision the session was adjourned. Shortly there after, Bill would put on his "decision" hat and meet once again with the inventor. With appropriate logic and sensitivity, judgment was rendered and a decision made about the idea. This process provided the inventor with a sense of satisfaction, even when the decision went against the project — a vitally important outcome for engendering continued enthusiasm and creativity.

Bill Hewlett was passing judgment, but Amy could use the spirit of this technique. Her first step is to recognize that Kotz's enthusiasm is very valuable to the project, even if it bugs her. As a veteran, she has on occasion worked with uninspired art directors who glumly create standard-looking artwork, which never makes for an exciting game. Good games have passion in them, and if she wants to make good games, she's got to understand and empathize with the passion, at least a little.

So her first reaction should be to the spirit of the sketch, not the problems with it. "I'm into drawing fairies! Back in art school, I drew these kind of wood-gnome-elf characters, kind of like this sketch, but more realistic-looking."

Kotz is curious. "Really?" he asks. "I'd love to see them! Were they photo-realistic, or more like Norman Rockwell fake-real?"

"Well, sort of like... hard to say. I have some photos of the canvas I'll bring in tomorrow if you want. So anyway, what kind of outfit were you thinking?" and Amy steers the conversation around to gathering more design input.

Once she's gathered an overall sense for Kotz's vision of the character, Amy can start exploring the technical issues of creating game art that works in the engine. For example, she might think aloud, "Hmm, so if we've got four frames of animation, we could do a little wing-flap bobbing cycle. But maybe we couldn't get an eye-blink in there because it would happen too often."

From these musings Kotz would learn what Amy's limitations are, giving him a chance to calibrate Amy's judgment to match his. "Hey Amy, wanna just skip the eye-blinks? It's not a big deal, right?" After a few rounds of this, Amy and Kotz will each become familiar with the other's style and where they disagree. If Kotz knows Amy's style, he'll trust her judgment and avoid micromanaging her in the areas where they agree — and he can keep control of the areas where they differ.

Problem 2: Art Direction

The second problem is the obvious one: the sketch doesn't say enough to build art. Kotz needs to provide a lot more information before Amy can build any production artwork. Of course, Kotz doesn't want to spend a week describing art that could be built in a day, so there's going to be a trade-off between thorough communication and time. Here are some common ways in which art directors communicate their ideas:

ARTISTIC REFERENCES. Torn-out magazine photos, scenes in movies, well-known artwork (for example, the Statue of Liberty) — a good reference gets an idea across quickly, but rarely matches the desired style exactly. Most art directors use them in combination or with limitations: "like this photo, but no neon signs and dirtier." The artistic reference can be a completely different subject from the planned artwork — for example, a mermaid drawing that has the right type of color saturation and detail for a fairy.

HAND-DRAWN SKETCHES. Character sketches, top-down scene layouts, storyboard cel-frames — they're all powerful methods of description. Often it's quicker to label special features than draw to draw them — for example, write "dirty leather belt" with an arrow pointing at the waist. Making quick, useful sketches is an art discipline in itself, so we'll leave that for a different article.

WORD DESCRIPTIONS. Stories and essays are also common ways to describe artwork — they are easy to compose, but leave a lot open to interpretation. Often, text descriptions are best for giving an overall sense for the character or scene, and will work best when combined with a few detail sketches. Word descriptions are good for technical issues, such as an exact list of a character's poses or the number of frames for looping animations.

REFERENCES TO ARTIST'S EARLIER WORK. This is a really powerful way to define style since the artist knows exactly how the artwork was created. This is one of the most powerful benefits of working with the same artists on multiple projects. Without that shared experience, it's often time consuming to convey something subjective such as style. If the director can get the artist's earlier work and can take the time to review the new art in terms of the old, the definitions are often very convincing.

ACTING. For describing animation, a quick and powerful method of communication is to actually act out the motion in person — a goofy strut, a wild swing, a sulky slouch-stance — all these motions can be acted out in seconds, whereas good sketches could take days.

Necessary Information

A good art director can creatively communicate a number of technical requirements as well. For 2D sprite art, here are some specific examples of what the art director should share:

EXPECTATIONS OF QUALITY AND STYLE. Artistic references tell the artist how much detail to create, as well as what kind of artistic style is sought.

TECHNICAL SPECIFICS. There's a host of inevitable limitations: dimensions of sprites in pixels, how your program recognizes transparent pixels, possibility of antialiased edges, number of colors (and if you are using a fixed palette, a copy of the palette), file format to deliver in, naming conventions, and so on.

CAMERA AND LIGHTING. What angle should these be drawn from: Overhead? Isometric? Side view? What background will they be viewed against? Is there perspective? Should there be bright, shiny highlights?

CHARACTER DEFINITIONS. We'll need a description of the unique attributes of each character. That first sketch is a

great start for character definition, but ultimately, we'll need more information: Man or Woman? Age? Culture (as indicated by clothes)?

ANIMATION. A list of required animations is a must. Art directors should also communicate the motion style that they expect. The technical issues usually rear their ugly heads here, too: number of frames available, type of transitions, and so on.

Ways to Direct Artwork

The methodology of art direction is pretty complicated in and of itself. Whole books have been written on the subject (see References). Still, there are some specific techniques that can help Kotz and Amy forge a more productive working relationship.

FRAMING. Framing is the concept of putting limits on the desired style, rather than naming the style itself. It's like drawing the space around an image, or "proof by negation" in math. For example, Kotz might tell Amy, "This fairy is fatter than Tinkerbell, but not as adult-looking. It's not a baby or a standard Christian angel, either." When used in addition to descriptions of the character itself, framing is a great way to rule out huge regions of style.

CONFIRM DECISIONS. After Kotz has tried to describe the concept, he wants to know if Amy really gets it or is just nodding along. Amy can demonstrate her understanding (and reveal areas she doesn't get) by offering some new design ideas that she thinks fit with Kotz's concept. For example, she might suggest that there be a fat little fairy and a tall thin fairy, and the two argue and tease each other. Kotz then can see that she's thinking of the fairies as having very clear, strong, goofy personalities, rather than being ethereal, ghost-like creatures.

There are some dangers with this approach. First, Amy has to expect that her suggestions will be shot down — if she has easily injured pride, she may not be willing to offer suggestions. Also, if Kotz thinks Amy is trying to take design control and doesn't understand that Amy's trying to prove her understanding, he's not going to be O.K. with her input. Obviously, Amy should preface the talk with a diplomatic comment, such as, "Let me see if I've got the idea. I'm not actually suggesting we build anything new, but..." It's also possible that Kotz will like Amy's ideas and give her more creative control than he was planning.

Our story ends happily. After Kotz and Amy have an hour-long discussion about fairies, Kotz leaves with the idea that Amy is into the concept but needs more details before she can actually build anything. Amy thinks that Kotz is kooky, but is open to the reality of making games and understands what she needs. Their future is all roses, and the world will soon be blessed with the fruits of their relationship: a game with great artwork. ■

REFERENCES

Art Direction for Film and Video by Robert Olson (Focal Press, 1993).

Sets in Motion : Art Direction and Film Narrative by Charles Affron, and Mirella Jona Affron (Rutgers University Press, 1995).

ARCHITECTING A 3D ANIMATION ENGINE

BY SCOTT CORLEY

It always helps to know what you're getting into. About three years ago, I was asked to volunteer my time at a junior high school (7th and 8th graders) to speak with groups of kids. They were having Career Day, and most of the people coming in to speak about their careers were parents of the children in school. I had attended this particular junior high school years ago, and a friend of mine who now works there contacted me a few days before Career Day to see if I'd come talk about working in the video game industry.

When I arrived at the school, I checked in at the principal's office, and amidst some confusion, they were able to tell me where I'd be presenting. I thought it was strange that they had chosen the girl's gym as my

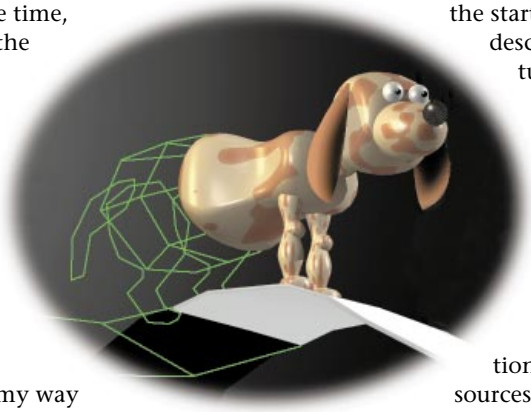
When Scott Corley was in junior high, he wanted to be a firefighter. Unfortunately, all he managed to become was vice president of software development at High Voltage Software. Express your disappointment at scottcy@ripco.com.

The interface to a 3D animation system can be very simple or very complex. If you've written one, you know what I mean.

. . . If you haven't written one, you probably will someday soon.

Here is a set of C/C++ interface guidelines for an animation engine.

presentation room, but it didn't concern me too much. I hadn't been in a junior high school in quite some time, and from the moment I walked in the front doors, I realized I was in a different, mostly shorter, world.



I made my way to the girls' gym. The gym teacher was there to greet me, and she explained how the day's schedule worked. Then she dropped the bomb. Nobody had told me this yet, but the reason I was asked to come in was because another presenter had canceled. This didn't strike me as horrific until I learned two more facts. The other presenter had canceled *after* the kids had chosen the careers they were interested in. The other presenter was scheduled to talk about sewing.

That day, I learned that 13-year-old girls who are interested in sewing and junior-high kids who are interested in video games are mutually exclusive groups. The talk was a disaster. I bombed. The discussion quickly degenerated to questions like, "Are you a surfer?" The only thing that saved me was the girls' gym teacher, who kept the group of giggling girls from exploding into full-on 13-year-old giggling girl anarchy. By the end of the day, the results were in. Everybody loved the firefighter (he brought in lots of cool equipment), and everybody liked the clown (everybody except, perhaps, the clown's kid). Nobody mentioned video games. So much for my first attempt at being a positive role model.

The moral of this story is: Know what you're getting into beforehand. Applied to 3D animation engines, this lesson dictates that you determine exactly what you want your engine to do before you begin developing it. If you don't have a good grasp of the requirements, your initial engine will have only the most basic features and won't be sufficient to support your game. You could find yourself adding various capabilities to the engine during the course of developing your game. Why not get it right from the start? This article describes a fairly full-featured interface for just such an engine.

Gathering Requirements for Your Engine

There are a number of informational resources and sources of inspiration to investigate before you begin developing your 3D animation engine:

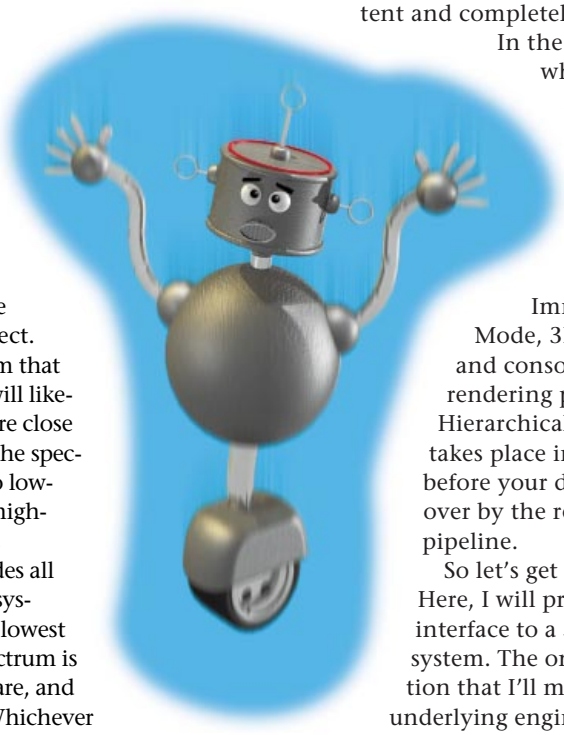
- If you've already written your own 2D or 3D animation API, consider what features you'll be able to add to it.
- Get documentation on commercial high-level 3D APIs that support animation. They'll inspire you to think in new directions and develop new features.
- Look at some other 3D game titles in development. What features do they use that you would like to implement in your next title?
- Discuss your ideas with someone who is knee deep in a large project. Their experience and perspective will undoubtedly expose ideas that are very useful and not obvious at the start of a project.

The 3D system that you start with will likely fall somewhere close to OpenGL on the spectrum of high- to low-level APIs. The high-level end of this spectrum includes all retained-mode systems, while the lowest level of this spectrum is you, the hardware, and an assembler. Whichever end of the spectrum you're on, however, you'll want to write your own animation control system.

At the high level of the API spectrum, there's usually support for animation, but it's unlikely that it will do you any good for the following reasons: performance, licensing fees, and feature set. Retained-mode APIs have a pretty consistent reputation for being slow and bulky. Some retained-mode APIs require licensing fees that you may not want to pay. And finally, animation support is often very basic.

For example, Direct3D Retained Mode allows you to define keyframes and the current display time for your animated object. You'll have to add a lot more code to make this system useful to you, and you have no control over important things such as the internal representation (storage size and accuracy) of the keyframes. Direct3D Retained Mode stores rotational keyframes as quaternions defined as four `float`s (16 bytes per keyframe). Using a custom 16-bit value in place of the 32-bit `float`s may provide you all the accuracy you want, and it will cut your animation data size in half. However, retained mode APIs won't provide you with this option.

At the low-level end of the spectrum, animation support is nonexistent and completely up to you. In the midrange, where OpenGL falls, animation support is also nonexistent. OpenGL, Direct3D Immediate Mode, 3Dfx's Glide, and consoles provide a rendering pipeline only. Hierarchical animation takes place in the steps before your data is taken over by the rendering pipeline.



So let's get down in it. Here, I will propose an interface to a 3D animation system. The only assumption that I'll make about the underlying engine is that you will implement the capability of interpolating between arbitrary keyframes.

LISTING 1.

```
class AnimatedObject {

// An interface to an animated object. Most comments are
// left out for brevity. See accompanying article for
// full descriptions of everything here.

// StandardTransitionTime is 1/10 second, 100 milliseconds
static const float StandardTransitionTime=100f;
// DefaultMsPerFrame is 1/30 second
static const float DefaultMsPerFrame=33.333333f;

// The intent is to provide an interface only, but some of the data
// mentioned in the article is presented below. Obviously, in a full
// animation engine implementation, there will be much more data
// in this object.

int currentAnimation;          // ID of animation playing now
float currentAnimationTime;    // current frame of current animation
int nextAnimation;             // ID of next animation to play, -1 if none set
float msPerFrame;              // current playback rate setting
bool loopCurrentAnim;          // true if current animation should loop

// Every animated object has to have a pointer to the animation data
// it is capable of playing. Actual implementation is up to you.
AnimationData *animationData;

public:
AnimatedObject(AnimationData *animDataIn); // constructor
void PlayAnimation(int animNum, float startTime=0f, float transitionTime=0f);
inline void PlayAnimationAtTime(int animNum, float startTime)
{ PlayAnimation(animNum, startTime); }
inline void TransitionIntoAnimation(int animNum)
{ PlayAnimation(animNum, 0f, StandardTransitionTime); }
inline void TransitionIntoAnimationAtTime(int animNum, float startTime)
{ PlayAnimation(animNum, startTime, StandardTransitionTime); }

void SetNextAnimation(int next);

void EnableLoopedAnimation();
void DisableLoopedAnimation();
bool IsAnimationLooped();

void PlayNextAnimation();
void TransitionIntoNextAnimation();

void StopAnimation();
void StartAnimation();

void SetMsPerFrame(float newRate=DefaultMsPerFrame);
float GetMsPerFrame();

void CaptureRotationChannel(int which);
void CaptureTranslationChannel(int which);

int GetCurrentAnimation();
float GetAnimationTimeInMs();
float GetAnimationTimeInFrames();
bool InTransition();
```

CONTINUED ON PAGE 30.

Defining Functions to Play Animations

The first part of our animation interface will deal with playing animations. With one very simple function, many different modes of animation playback are covered.

For all code listings, I'll use a C++ style that assumes we have a class that implements this animation interface. Every function interface listed in this article is actually a member function (or method) of our **AnimatedObject** class (Listing 1). If you're using C, just mentally insert a pointer to the animated object **struct** as the first argument of each function, then grumble about the lameness of C++ for a bit. Likewise, you can use your imagination to eliminate the default arguments used in the examples if you are using a strictly C compiler that doesn't support default arguments.

The **PlayAnimation** function,

```
void PlayAnimation(int animNum, float
    startTime=0f, float transitionTime=0f);
```

gives us the very basic ability to start playing any animation that our character is capable of playing, as defined by **animNum**. It also allows us to supply a starting point, **startTime**, within that animation; depending on the game, there may be many occasions when you want to start playing an animation "in progress" by skipping a few milliseconds of that animation. Finally, we allow ourselves the ability to define the transition time between animations with the **transitionTime** argument. This time can mean different things to different people. For some, it can define the number of milliseconds inserted between the previous animation and the one that you are attempting to play. For others, it can define a period of time during which both animations are played and averaged together. Either way you look at it, this time will normally be a standard value that makes the transition between animations look good. The other common value for **transitionTime** will be zero, indicating that the next animation must start immediately without any interpolation at all.

From **PlayAnimation**, we can easily come up with a few commonly used variations to define within our animated object class.

LISTING 1 (CONT. FROM PAGE 28).

```
bool IsAnimationLooped();

int GetNextAnimation();
void GetGlobalPosition(int joint, float &x, float &y, float &z);
void GetCurrentVelocity(float &x, float &y, float &z);
void SetCurrentVelocity(float x, float y, float z);
float GetAnimLengthInMs(int animNum);

};
```

LISTING 2.

```
inline void PlayAnimationAtTime(int animNum, float startTime)
{ PlayAnimation(animNum, startTime); }
```


LISTING 3.

```
inline void TransitionIntoAnimation(int animNum)
{ PlayAnimation(animNum, 0f, StandardTransitionTime); }
```

LISTING 4.

```
inline void TransitionIntoAnimationAtTime(int animNum, float startTime)
{ PlayAnimation(animNum, startTime, StandardTransitionTime); }
```

LISTING 5.

```
void EnableLoopedAnimation();
void DisableLoopedAnimation();
bool IsAnimationLooped();
```

The function in Listing 2 will start a particular animation immediately. There will be no transition between the currently playing animation and the animation specified in this function call. The starting point of the new animation will be **startTime** milliseconds into the new animation.

The **TransitionIntoAnimation** function in Listing 3 will be the most frequently used function to start a new animation. All you need to specify is the new animation to play. The transition period between the currently playing animation and this new animation is set to the **StandardTransitionTime** (a **const** defined in a header file, typically equivalent to two or three frames' worth of animation). This function provides the smooth transitions that you'll normally want to see between animations.

The **TransitionIntoAnimationAtTime** function in Listing 4 will transition (interpolate) into a new animation and will skip the first few milliseconds of that new animation as specified by **startTime**.

All three of these functions are just convenient ways to call **PlayAnimation**; they are all ideal candidates for inlining. Each function states exactly what its purpose is, so you can tell what's going on in the code without trying to interpret individual arguments to **PlayAnimation**.

What happens when an animation is finished playing? Our animation engine has two options. It can loop the current animation, or it can transition into the next animation. We need a way to define what the next animation is. This can be achieved with the following function:

```
void SetNextAnimation(int next);
```

The **SetNextAnimation** function can be called at any time, but it is most appropriate to call it immediately after playing an animation. That way, you know exactly what will happen after an animation is finished even if you don't get around to playing another animation. Our engine can make some fairly intelligent decisions regarding looping animations at this point. As soon as an animation is played with any variant of **PlayAnimation**, assume that the animation will be looped. Then, as soon as a next animation is defined via **SetNextAnimation**, disable the looped status of the current animation (otherwise the next animation would never be reached), and assume that the next animation will be looped. If these assumptions ever change, we'll conveniently have functions to manipulate or examine the looped status of the currently playing animation.

The functions in Listing 5 do exactly what you would expect. If the currently playing animation is looped (recall that our engine will automatically loop any played animation), you can change that status with a call to **DisableLoopedAnimation**. If you then change your mind, you can re-enable looping with **EnableLooped-Animation**. **IsAnimationLooped** returns the current status.

What would happen if you played an animation, set the next animation, and then called **EnableLoopedAnimation**, like this?

```
// Interesting code snippet
PlayAnimation(animNum, startTime);
SetNextAnimation(next);
EnableLoopedAnimation();
```

In this case, the initial animation will be looped. When you call **SetNextAnimation**, the playing animation will cease to loop. When you re-enable looping on the current animation via **EnableLoopedAnimation**, the next animation that you set will never be played. This is a perfectly reasonable sequence of actions that could arise in your game, and no harm will come of it. What happens if your next animation is playing looped, and you disable its looping? When that animation ends, your engine won't know what to play. This case can either be considered a bug (an error message appears with something clever such as "Should never get here!"), or you can define a default animation for each animated object that is played and looped when no more animations are instructed to play.

You can exert even more control over what happens with the next animation to be played. There will be times when you want the currently playing animation to end early, but you don't yet have another animation to play. Usually, the animation that was set with `SetNextAnimation` is a safe follow-on to the current animation (otherwise you wouldn't have set it as the next animation). You can skip directly to the next animation by calling either of these functions:

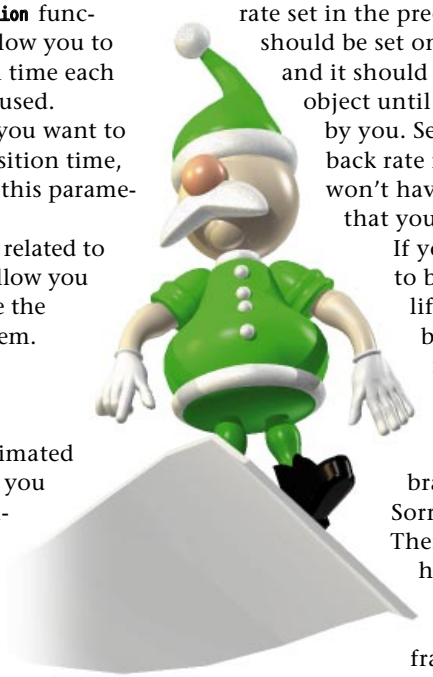
```
void PlayNextAnimation();
void TransitionIntoNextAnimation();
```

`PlayNextAnimation` will start the next animation immediately with no transition. `TransitionIntoNextAnimation` will start the next animation with the default amount of interpolated transition frames. You could add a `transitionTime` parameter to the `TransitionIntoNextAnimation` function, which would allow you to specify the transition time each time this function is used. However, in general you want to use your default transition time, so don't bother with this parameter unless it's a must.

The final functions related to playing animations allow you to temporarily disable the entire animation system. They are

```
void StopAnimation();
void StartAnimation();
```

If you want the animated object to freeze, or if you want to take algorithmic control over the animation, then you need to temporarily disconnect animation playback. `StopAnimation` stops animation playback in its tracks. Any updates to the current animation will be ignored. Even calls to `PlayAnimation` should be ignored until the animation is re-enabled with a call to `StartAnimation`.



Playback Speed

The animations that you play back with this interface will have keyframes, and those keyframes will have timestamps that define the animation's actual playback speed. Those timestamps will most likely be in terms of frames, where a frame is about 1/30 second. What if you want to play an animation back at a different speed? Give yourself an easy way to do this with functions like these:

```
void SetMsPerFrame(float
newRate=DefaultMsPerFrame);
float GetMsPerFrame();
```

The preceding functions allow you to set "milliseconds per frame" to define the playback rate of an animation. If your animations were originally 30 frames per second, the `DefaultMsPerFrame` value about 33.33. By calling `SetMsPerFrame` with a value of 66.66, you cut the animation playback rate in half and everything will play back slower than normal. Smaller values will make animations play faster. Note that the playback rate set in the preceding functions should be set on a per-object basis, and it should stay set for that object until it is changed again by you. Setting a global playback rate for all objects won't have the flexibility that you will need.

If you're accustomed to basing your entire life around a vertical blank interrupt, the idea of setting 33.33 milliseconds per frame is going to hurt your brain in some way. Sorry about that. There is some overhead to using milliseconds instead of an integral frame count, but it's minimal on today's machines. Basing everything on milliseconds rather than frames makes more sense if you ever plan on having the game run on more than one machine configuration. For consoles, there's a discrepancy between PAL (50 FPS) and NTSC (60 FPS). On a PC, the frame rate you can to achieve will fluctuate according to the performance of the machine on which your game is running. Basing everything in your game on milliseconds instead of frames is hard to get used to for some people (including me), but it's worth the effort.

Capturing Channels

An advanced feature that you may want to add to your animation engine is the ability to capture a channel. If you have a hierarchical animation system, you will have multiple channels of data for each animation, such as a root translation channel and a joint rotation channel for each joint in the hierarchy. There may come a time in your game when you want to take algorithmic control over one part of the character's animation and allow the rest of the character to animate normally. For example, you might want your character to look in a certain direction while running. To do this, you have to capture the channel for the neck rotation.

Capturing the channel itself is easy if you have a method of identifying translation and rotation channels by number. To accomplish this, you should add the following functions to your engine:

```
void CaptureRotationChannel(int which);
void CaptureTranslationChannel(int which);
```

The numbers that you use to identify channels should correspond to the numbering or ordering convention of your 3D animation tools. Your engine will respond to this capture request by simply stopping the animation for that channel only. For example, you may have 16 joints in a model being animated with rotation information every frame. Capturing joint 10 means that joint 10 will no longer be updated with rotation information every frame, but the other 15 joints will continue to be updated.

Once a channel is captured, you can do with it what you like, safe in the knowledge that changes you make to a captured joint's rotation or translation won't be clobbered by animation data. How you actually manipulate that joint depends on how your animation system is implemented.

Determining the Engine's State

As your animation engine grows in complexity, it becomes increasingly important to know about the state of the engine at various times during game execution. The solution, of course, is to create functions that retrieve this state information. Let's look at some of these.

LISTING 6.

```
int GetNextAnimation();
void GetGlobalPosition(int joint, float &x, float &y, float &z);
void GetCurrentVelocity(float &x, float &y, float &z);
void SetCurrentVelocity(float x, float y, float z);
float GetAnimLengthInMs(int animNum);
bool IsAnimationLooped();
```

Information regarding the current animation being played is certainly important. To identify the currently playing animation by number, use this function:

```
int GetCurrentAnimation();
```

To return the current position in the animation being played relative to the start of that animation (this value will continually reset to zero for looped animations), use:

```
float GetAnimationTimeInMs();
```

Similarly, you can return the current position in the animation being played relative to the start of that animation in terms of frames with

```
float GetAnimationTimeInFrames();
```

Why create two functions that return the same information in different formats? Recall that the playback speed of an animation can be changed. In some cases, you might be interested in knowing whether a particular frame of an animation has passed yet. Using `GetAnimationTimeInFrames`, you can determine this information regardless of the animation's current playback speed setting.

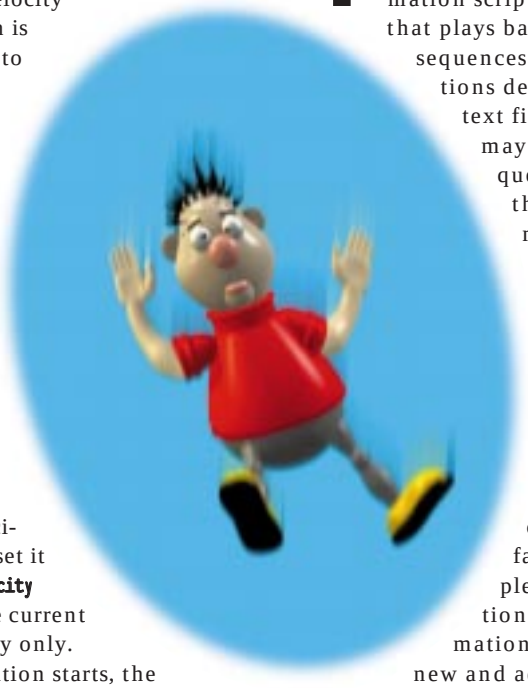
The `InTransition` function should return `True` if the animation engine is currently interpolating between two different animations:

```
bool InTransition();
```

The functions in Listing 6 provide you with additional state information that you will likely need. The first function, `GetNextAnimation`, will tell you what animation is set to be played next.

Sometimes, you need to know very specific information about the current state of the animation, such as the global position of a particular joint. Using your joint numbering convention, you can ask for the position of any joint in terms of world coordinates. In this example, the `GetGlobalPosition` function returns the x, y, and z values individually, but you'll probably use your own vector type here. The global position of any joint is essential for collision detection.

The velocity of an animation is an implementation-dependent concept. If you've extracted a velocity from your animations and stored this velocity with the animation, you can use `GetCurrentVelocity` to find out what the current velocity is. This information is useful when trying to predict when an animated character will reach a certain location, for example. The corresponding `SetCurrentVelocity` can come in handy if you've determined that an animated character is moving too slowly to get where it needs to be. Get the velocity, scale it up, and set it again. `SetCurrentVelocity` should override the current animation's velocity only. When a new animation starts, the correct velocity for that animation should be used. Note the `SetMsPerFrame` function that I already discussed will affect the velocity as well as playback rate, so if a persistent change in speed is what you're after, use `SetMsPerFrame`.



Often, you'll need to know how long an animation will take to play back. `GetAnimLengthInMs` will return that information for any animation that a character is capable of playing. This information can be useful in hundreds of situations. This call should take into account the current milliseconds per frame setting.

In addition to the length of a particular animation, there is a long list of other information that you may want to know before playing an animation. For example, you might want to know which frame of an animation has the largest z translation from the starting position. For custom information such as this, write a routine that can query individual frames of individual animations. Note that in a keyframed, interpolation-based system, the frame that you inquire about might not physically exist. In this case, the query will have to create the interpolated frame in order to return the information that you're after.

And More

Perhaps you have an animation scripting system that plays back sequences of animations defined in a text file. Or you may need to queue up more than one animation at a time. These needs are simple extensions to the system presented in this article. If you start out with a fairly complete foundation to your animation system, new and advanced features will come easily.

Many ideas are presented in this discussion of a theoretical 3D animation system interface. All of the ideas are from real-world examples. Nonetheless, your real world will always be different. The entire set of features that you need can only be determined by you; hopefully, by now you are thinking well beyond the ideas presented here and picturing your ideal animation system. And if anyone asks you to speak to kids about any of this, walk away. Just walk away. ■

Acknowledgements

The author would like to acknowledge fellow High Voltage guy Dwight Luestcher for his part in developing the concepts discussed in this article.

The Art of QUAKE 2

by Paul Steed

As in nearly all aspects of the company, the art department at id Software runs on pure synergy. That is, no real distinct lines of responsibility exist among us three artists (other than the fact that fellow artists Kevin Cloud and Adrian Carmack — along with programmer John Carmack — also happen to own the company) because we work together so well as a team. Sure, we each have our particular strengths and weaknesses, but in the end, any of us can do any aspect of the art needed for the game. The strength of the three-man art department at id comes from experience, hard work, and talent. Adrian Carmack, Kevin Cloud, and I created the art for QUAKE 2 (and the upcoming mission pack), and this article reveals how it all came together.

Production Design

Our team begins the art process with a solid production design and tons of sketches. This is the basis upon which we build models, create map textures, or skin characters. We contracted a very good production design artist to help with the environment concepts for QUAKE 2, but for the most part, conceptual art and production design comes from one of the three artists here at id. Although we had to design the art based on our game design ideas, our main goal was simple: "Make it cool." That pretty much summed up our design philosophy.

Our team went for a grungy, sci-fi look for QUAKE 2, so the enemies are barbaric races of miscreants sporting an array of cybernetic prostheses. The rationale behind this design was that the inhabitants of Stroggos (the planet where the game takes place) aren't much into aesthetics and are purely a warlike, spartan race. The good guys were designed with a futuristic military look. Weapons, armor, and other aspects of the Terran Coalition of Man are easily recognizable and not too far removed from contemporary military and weapon design. Our intention was to spare the player the effort of stretching his or her imagination too far in pondering the purpose of an object. Is this a weapon in my hand, or is it a bread-maker?

Noticeably missing from QUAKE 2's production design was any form of demonic symbology. This wasn't due to any political or publisher-induced corporate pressure; it was because it didn't fit the theme of the game. If the Strogg had developed into a magical/netherworld-dependent race of conquerors, we would have had plenty of demons running around doing demon-type activities. QUAKE 2 is about kicking some alien butt on their turf and ensuring mankind's future survival.

After attending 22 different public schools spanning five states, serving his country for six years, and bouncing around Europe for three more, Paul Steed taught himself how to create art on a computer at the University of Origin (Austin branch). After almost finishing up a 4-year degree in low-polygon model creation and animation and cinematic techniques, he spent time at Iguana Entertainment and Virgin Interactive Entertainment before being recruited to the Mt. Olympus of PC software development, id Software. He doesn't plan on going anywhere else anytime soon.

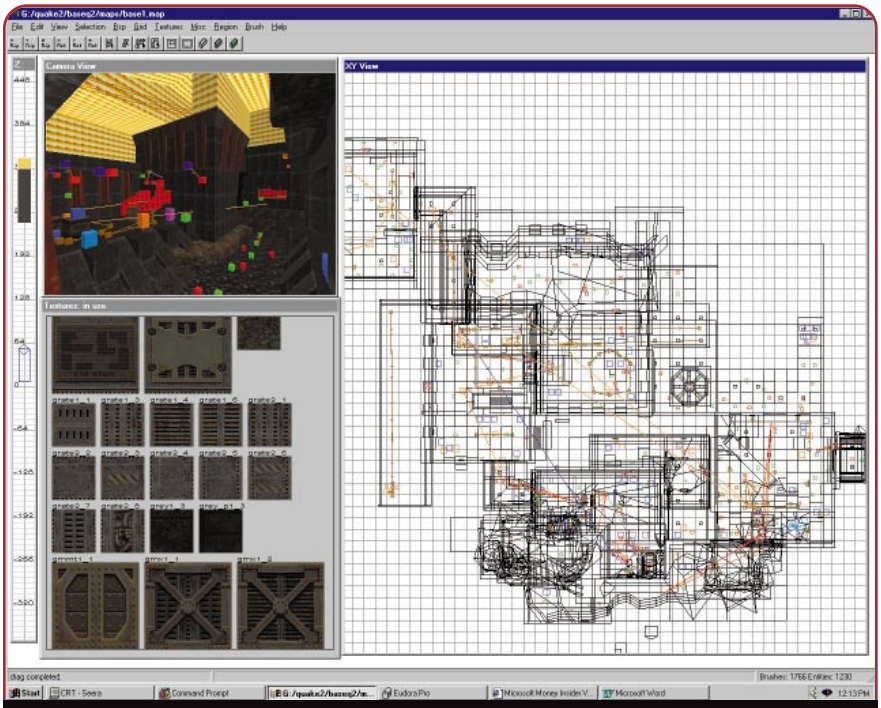


FIGURE 1. *id's in-house level editor.*

Level Texture Maps

Although technically there are only three artists at id, an argument can be made for including the level designers in the pool of pixel and vertex pushers. The designers make maps, and in order for them to build the world, they need plenty of building material. We used an in-house editor (designed by John Carmack), which allowed level designers to take our textures and turn them into geometric shapes. The level designers then used these shapes to fill in the architecture (Figure 1). We call these shapes "brushes."

Typically, the process of handing the level designers the textures that they needed for a map went like this. First, we artists created a series of "base" textures — generic textures with a certain look that could be reused throughout different areas of the game. We also created textures specific to certain areas as dictated by the design document, such as the "mines" texture set or the "factory" texture set. All textures were required to be sized in a power-of-two (for example, 8 pixels × 8 pixels, 16×16, 32×32, 32×64, 128×128, 128×64, and so on) (Figure 2).

All of the in-game textures in QUAKE 2 were created using Electronic Arts' venerable Deluxe Paint, and for some animations we also used Kinetix's Animator Pro. You may chuckle at our choice of Deluxe Paint, as the program has been around for ages. But for static manipulation of pixels using a 256-color palette, there really is no substitute. Sure, Photoshop and other programs designed to work in true color can be used to do 256-color tweaks, but Deluxe Paint is a worthy tool.

The reason QUAKE 2's textures turned out so well is due to our hand-drawn approach to creating them. Using what's called "nearest pixel" methodology, we ensured the integrity of each texture by paying special attention to the proper amount of antialiasing. Sometimes, a stippled or dithered effect is appropriate, but simply rendering a scene in a 3D package and slapping it into a game without modifying the image will often result in "pixel swim." While massaging an image pixel by pixel is painful and time-consuming, it's a necessary process. However, as 3D accelerator cards become more popular (and in some cases, mandatory), bitmap interpolation will address the problems associated with pixel swim.

Skinning Characters

To skin our characters (skins are textures that are projected linearly onto a character's model), we had to warp the models' base frames in order to get the maximum amount of coverage (a base frame being a reference-only state of deformation for the generation of the texture map) (Figures 3a and 3b).

To help us artists skin characters, John Carmack wrote a tool (in a weekend, no less) called Texpaint, which

FIGURE 2. *One of QUAKE 2's .LBM texture files, created in Deluxe Paint.*

FIGURE 3a. *A character skin texture map.*

FIGURE 3b. *And the skinned character.*

allowed us to paint directly onto a mesh, much like the commercially available tool Positron's MeshPaint. Texpaint proved to be invaluable. Though fairly simple, the tool nevertheless featured unlimited undo capability, showed model and texture lines, and let us select any pixel resolution for the texture sheet. It also gave us the ability to generate a triangle-unique multicolored "start" skin for clarity and identification of possible problem areas (Figure 4a).

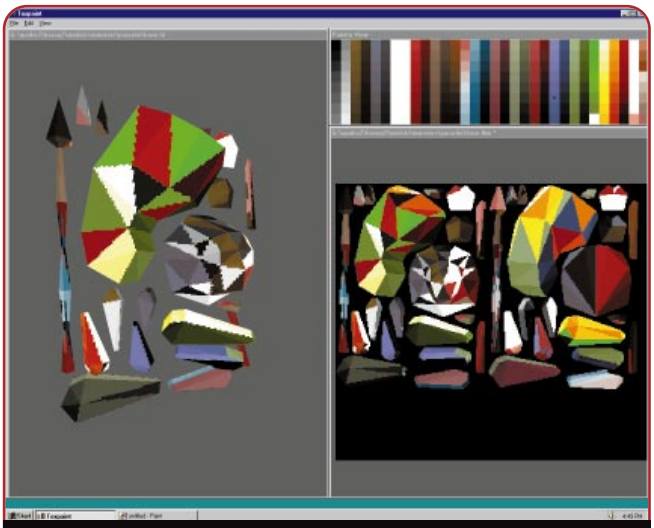


FIGURE 4a. *John Carmack's character skinning utility, Texpaint.*

Using Texpaint to create a skin was a simple affair. First, we'd load up the base frame and click the New Skin menu item, which generated an .LBM (Deluxe Paint's native file format) for the character on which we wanted to paint — the .LBM's resolution was based on the mapping coordinates of the loaded object. Then, we'd bring in any other non-base frame's object file from the object's directory, retaining the previously generated base texture (Figure 4b). Texpaint would then help us take the model from its deformed base frame state to a polished, finished object (Figure 4c).

As we'd draw on the .LBM object, the changes to the skin appeared simultaneously on both the texture page and the skinned object itself. This method worked especially well when we used Texpaint and Deluxe Paint in conjunction with Animator Pro. Our process consisted of marking out certain areas on the skin, using either Deluxe Paint or Animator Pro to refine these areas, and then reloading the skin into Texpaint for further refinement. All of the artists on our team had at least two computers (a Windows NT system and a Windows 95 system), so using the Windows NT-based Texpaint and the DOS-based Deluxe Paint in tandem wasn't really a problem because we could rely on our network to transfer the files and our twin 21-inch monitors for instant feedback. Once textures were completed, they were handed off to the programmers, who converted them from .LBM to .PCX format for better consumption by the game engine.

Inanimate game objects, including the view-model weapon that characters used, required only one skin since these objects didn't change states during the game. (View-model weapons are those that you see in your hands while playing the game, and world-model weapons are those that you run over to add to your inventory.) The monsters and player characters, however, required an additional "pain" skin to convey that the creature or player was hurt. This different texture wasn't sector-based in any way. We merely switched out entire skins when a character's hit points reached a critical level.

View Screens, Option Screens

The look for the main-view and option screens in QUAKE 2 were designed to fit the basic militaristic theme of the game. We chose a clean and uncluttered look for the status bar for simplicity's sake. When we released Q2test (the public beta version of QUAKE 2) on the Internet, we featured a face at the bottom left of the screen (likely familiar to QUAKE and DOOM players) that reflected the damage state

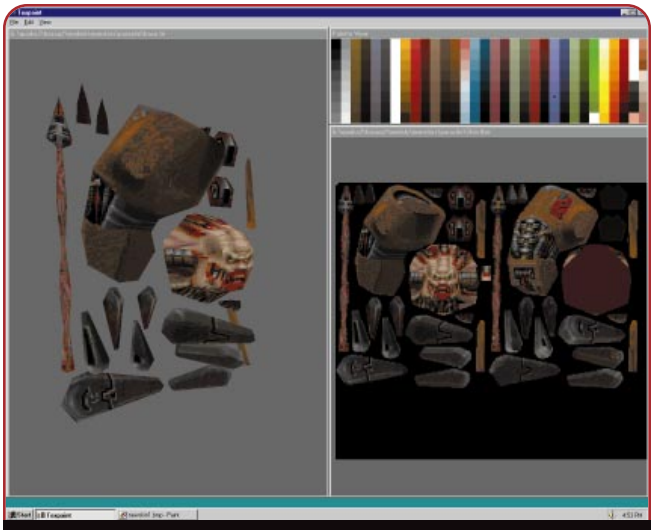


FIGURE 4b. *Bringing a non-base frame's object file into Texpaint.*

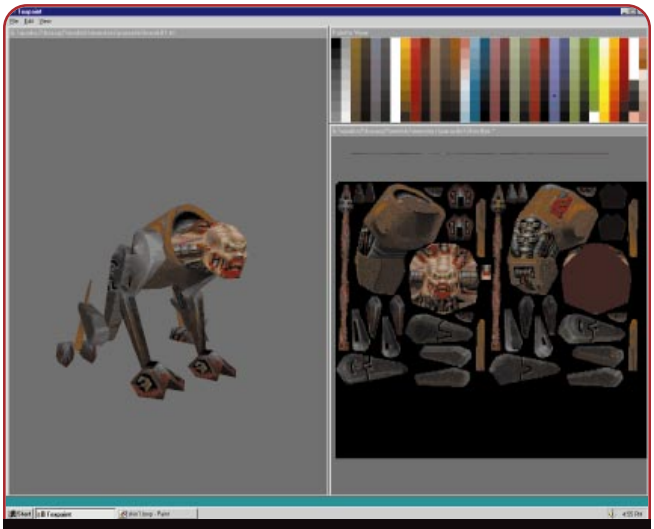


FIGURE 4c. *The skinned character after refinement in Texpaint.*

of the character and even indicated from which direction the player was being hit. Unfortunately, since players can choose from about 30 different deathmatch characters, we decided that implementing an animated face for each deathmatch character would be too time-consuming — we decided to use the universal red cross on a white background instead (Figure 5).

Although no faces are displayed on the status bar during game play, the player does get a snapshot of the chosen character in the multiplayer setup menu (Figure 6). A rotating model of the chosen character also allows casual perusal of the model and skin. While we would have liked to use these same snapshots in the status bar during game play, it wasn't possible due to their resolution. The portraits of the characters in the multiplayer setup menu are 32×32 pixels; the highest possible resolution for any graphic in the status bar was 24×24. It all came down to the fact that the 32×32 versions were done first, we didn't have time to do secondary versions of them, and we didn't feel that 24×24 was enough pixels to do justice to the different faces. So we opted for the red cross.

QUAKE 2's view and option screens were created last and constructed with functionality and practicality in mind. We were constrained to these screens by code that was already in place from the previous game. In addition, while we would have loved to have had all kinds of tricky animated buttons in the front-end menus, why make the player run through a maze of information just to change an option? Time and functionality concerns dictated QUAKE 2's spartan interface.

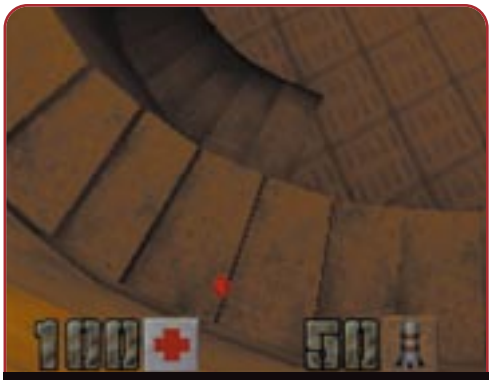


FIGURE 5. *id's artists chose to represent health as a simple red cross.*

Modeling

Basically, two types of models exist in QUAKE 2: Alias models and B-models. Alias models were created using Alias|Wavefront's PowerAnimator on an SGI (an Indigo 2 Extreme with 256MB RAM). B-models are "brush models" created by level designers from the previously mentioned textures. Another way that we gave the designers working material for their level creation was in the form of an internally-developed tool called 3DS2MAP. This little utility took nonconvex shapes built in Kinetix's 3D Studio R4 and converted them into brush models that could be used in either QUAKE or QUAKE 2 levels. 3DS2MAP helped overcome some of the deficiencies of the designers' in-house development tool (the QUAKE Editor), such as its inability to create perfect spheres, domes, or weird geomorphic shapes such as boulders or rocks. Tim Willits, the lead designer on QUAKE 2, gave me the strangest look when I made him a half-dozen rocks one time in about five minutes; he said it would have taken him days to do that in their editor.

Alias models were used for characters and objects in the game as well as the view model and world-model weapons. We also used Alias models for miscellaneous map-specific objects, such as radar dishes and the ships that fly by occasionally. We used 3D Studio R4 for the bulk of the character modeling because once the normals are reversed, 3D Studio files can be exported as .DXF files (the AutoCAD file format) and easily imported into PowerAnimator.

FIGURE 6. *The multiplayer setup screen gives players a rotating view of their chosen character.*

If you're wondering why we didn't just use PowerAnimator's modeling capabilities for every object, the answer is me. Kevin Cloud actually did use PowerAnimator exclusively. However, I was responsible for most of the models and I used the product that I know best: 3D Studio R4. Simply put, I don't care for PowerAnimator's modeler. I know 3DS4 like the back of my hand and treat it as my 3D sketchbook. I can complete a model in about the same time that it takes to work up the same design on paper.

Objects' face counts were dictated by the "less is best" doctrine (that is, be as economical as possible with model polygon counts). The creation of all the models was, however, an iterative process, and optimizing them was an art form unto itself. For example, Figure 7 shows a high-resolution version of the gunner that I created for an Australian magazine cover, along with the version of the same character as it appeared in the game.

CHARACTERS. Our goal was to keep basic enemy characters to 600 faces or less, although some reached as many as 700. Most were in the 400 to 500 range. Boss characters, due to their uniqueness and the fact that they were often alone on their level, ranged from a hefty 1,500 faces to a whopping 2,500 faces for the final boss Makron (astride his mount, Jorg). The player character was exactly 600 faces, but because we did the weapon as an overlay, you could actually say the male and female player characters were around 750 faces each.

VIEW-MODEL WEAPONS. When you play QUAKE 2, the weapon model that you see in your hands is slightly to the right or left. This is because of the way that we optimized the view-model weapons (Figures 8A and 8B). The view-model weapons are the most optimized models in the game due to their visually restricted and visually predictable nature. What you'd see if the weapons were slid over to the middle would be hardly recognizable as a weapon, because the side view shows more of the texture mapping details. Keeping the weapons (and the hands holding them) under 400 faces was no easy task. Not only did the weapon and

FIGURE 7. *The low-polygon version of the gunner (right-hand side) weighs in at around 620 faces. The high-resolution version (left-hand side) is about 20,000 faces. When optimizing, Paul Steed used an external plug-in for 3D Studio R4 called Optimize, which gave him a head start on a super-dense mesh. Once he'd reduced a model down to about 1,000 faces, he would finish the optimization by hand. Nearly all tools of this kind simply smooth off and merge faces based on their relational or incidental angles, and often some manual work is required by the modeler to polish off the job.*

hands have to look realistic, but in most cases, had to animate as well.

Creating the view model so that it appeared correctly in the world was sort of tricky. Because the game engine gives you a ninety-degree field of view (FOV), objects that are close up are severely skewed and seem stretched out. To adapt to this FOV limitation, we created the weapon models and then viewed them in a camera window approximating the same view of the game. The models also had to be squashed somewhat to compensate for the stretching effects on nearby objects in the FOV. Only after quite a bit of tweaking did we achieve the correct results. Once we were satisfied that the weapon looked good from the proper angle, that version was saved and the arduous task of optimizing it down to more digestible measurements began. In my opinion, the weapon view models were probably the most challenging and most satisfying of the models done for QUAKE 2.

GENERAL OBJECTS. General objects such as ammo boxes, armor, weapons, or anything that you can you pick up were also created with the "less is best" philosophy (Figure 9). These objects rarely surpassed 200 faces. The subtle pulsing glow around the objects is for the gaming aspect of the artifact and is understandably necessary to ensure its visibility.

The modeling in QUAKE 2 was done with the same attention to detail as all other parts of the game. We wanted to make visually interesting characters and objects that fit within the plot and design of the game. We put as many characters and objects into the game as the single CD and time would permit. The characters and objects that didn't make it into QUAKE 2 will most likely rear their low-polygon, ugly heads in the upcoming mission pack that we're working on.

Animation

All of the in-game, nonrendered animations in QUAKE 2 were done in PowerAnimator. We feel PowerAnimator is simply the best program for character animation available. The strength of PowerAnimator for doing character animation lies in the amount of control you have over each vertex. A tool that provides subtle degrees of influence over individual vertices makes a big difference in the realism of a bicep or elbow joint that maintains its integrity as it flexes. If your program doesn't allow you to control elements at the vertex level (as opposed to a tool that merely assigns influence to an area of vertices), you'll run into problems sooner or later.

CHARACTERS. The following is QUAKE 2's general character animation process:

- Create the character or import it from 3DS4 into PowerAnimator.
- Build a skeleton inside the mesh to support its animation.
- Attach the mesh to the skeleton and name the clusters accordingly.
- Assign an appropriate amount of influence over the vertices or groups of vertices (clusters).
- Flip any edges that have to be changed in order to better support

FIGURE 8A. *The view-model mesh of a machine gun.*

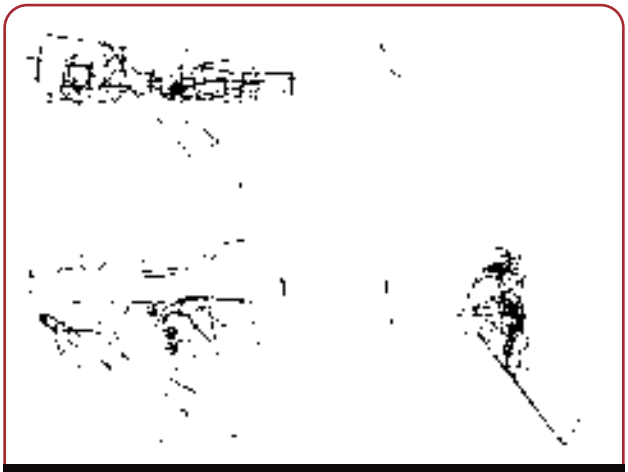


FIGURE 8b. *Several orthographic views of the same view-model mesh.*

the animation (especially around the hips or shoulders), thus avoiding any unwanted "crimps" or other ugly deformations.

- Create any applicable inverse kinematic (IK) chains (I only used IK for legs or arms if the creature used them in locomotion).
- Make a copy of the character's feet and save them as templates for future reference.

Setting up a skeleton and mesh so that they would work well together usually took about a day and was the single most time-consuming and arduous part of the character animation process. Associating over 300 vertices into groups, naming them, and making sure that they were linked to the right joint made me sometimes wish we had an intern. If the process isn't done perfectly, you'll regret it later. One of the problems we faced — even with PowerAnimator — was that even though you can set a keyframe for an IK handle and make the body sway or lead over time, the feet at the bottom of the IK chain will turn and twist. That's why I always made copies of the character's feet and saved them as templates. These guides ensured that there was no skating effect by characters as they walked.

The actual animation process is my favorite part of my job. Early on in the project, we debated whether to animate at 10 frames or 20 frames per second. John Carmack had linear interpolation working within the engine from Day One, so we knew that the faster machines would make our animations as smooth as silk regardless of the frame rate. We ended up going with a 10 FPS base for several reasons. First, it meant less data storage because we weren't going to use an in-game skeletal animation system. Instead, we used the QUAKE method, which tracks the vertex position based on differencing the position of the vertices frame by frame. The second reason was that even our slowest target machine could play back the animations at 10 FPS without a problem.

Because we went with 10 FPS instead of 20 FPS, we never even considered using motion capture to animate our characters. There was no point — what good animator can't animate at 10 FPS? Just as all the textures and skins are hand drawn, the animations were also created by hand without secondary data. When we dive into developing Trinity, however, we definitely will explore the benefits of motion capture.

The basic animation set of the characters generally consisted of an idle animation, an idle animation in which the character did something interesting at random intervals (such as scratching a certain anatomical part), a walk, a run, several attacks, several expressions of pain, several deaths, a duck or crouch, and a blocking animation (which didn't make the final cut). This is a very generalized list, though, and animations sometimes varied drastically from character to character. Typical characters had between 250 and 500 frames of animation total.

One regret we had with regard to our animations was not putting all of the animations for a given character into one file. When you have a run anima-

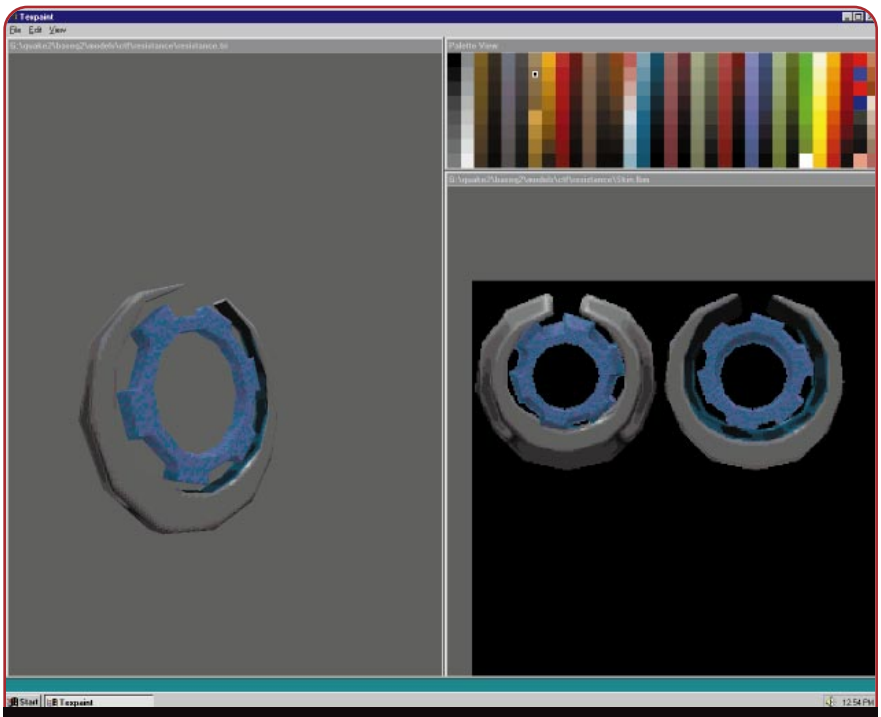


FIGURE 9. *Random objects in QUAKE 2 were created with the same care as the main characters.*

tion, a stand animation and three pain animations saved as different files, making changes to a character's main mesh necessitates changing a sequence of files, rather than just one. For example, adding the flag for the capture-the-flag multiplayer mode would have been much easier with this method. Instead, the flag had to be grouped to each of the subsequent character animations, making the job take about one-and-a-half times longer that it should have.

Once a character's animation suite was complete, the arduous process of saving the frames began. Each frame of animation had to be saved as a .TRI file in order for the mesh to be added to the game. We would go through each animation file (in this case, for a character's pain animation) and save frames as a sequence of files such as PAIN101.TRI, PAIN102.TRI, and so on. Each character had its own directory, so we used the same naming convention for all characters. A spreadsheet containing all of the salient data about a character's animation set (file names, offsets, and general action description) was also maintained by the art team. This proved extremely helpful down the stretch when the animations were converted to .DLLs, and on those occasions when programming data mysteriously "disappeared."

VIEW-MODEL WEAPONS. Animating the view-model weapons was a lot trickier than the animations for the characters. The models were so optimized that a great deal of care had to be taken to animate them. Early on, the weapons were higher in view space to show off more geometry. This approach blocked too much of the player's view, so we lowered the weapon, which revealed even less of it and allowed us to optimize the models even more. For example, initially the chain gun had a base and some trigger buttons on top that corresponded to the world chain gun model. These features disappeared after we lowered the weapon and pulled it closer to the player's view.

Making the weapon appear as if it was being drawn and adding idle or fidget states to the weapons models was primarily John Carmack's idea. He wanted a little more realism in the weapon switching and weapon animations (as opposed to the instant weapon switch trick used in QUAKE), so we added finger taps and weapon readjustment animations .

We tried to add muzzle flashes but they didn't quite work out, so they didn't make the final cut. Although I'm personally dubious as to the effectiveness of any type of polygonal representation of fire, smoke, or any other sort of incendiary effect, Kevin and Adrian pulled off these effects very convincingly. Given more time, I'm sure we would have found some way to make a convincing flash effect or shell ejecting animation for the weapons. I guess we'll have to wait for Trinity.

As in all things id-like, an attempt was made to infuse attitude into all the animations to give an identity to the characters and weapons. Small details such as painful deaths, humorous taunts, and tapping fingers are examples of game elements that we wracked our brains over. I believe that our meticulous approach made the animations more memorable and resulted in a more enjoyable game playing experience. ■

Implementing a Texture Caching System

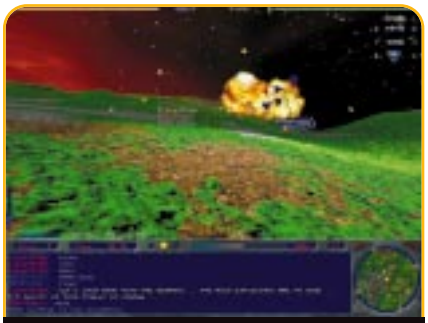
by Jonathan Blow

Texture caching systems are designed to overcome the texture budget limitations of 3D games. Only the textures required to display the current scene are held in RAM. When new textures need to appear in the scene, they are loaded from a larger and slower repository, or they are dynamically generated.

For example, textures can be pulled from disk into system RAM or downloaded from system RAM into the video RAM of a 3D accelerator. Textures can be dynamically generated by combining illumination maps with unlit source textures.

QUAKE was one of the first games to implement a texture caching system that interacts closely with the 3D pipeline to cache graphics in an efficient manner (see References). DOOM cached textures as well, but its system was more of a solid-state approach, as was the data caching scheme in the 2D side-scroller ABUSE. The source code to both ABUSE and DOOM is now available; see the References at the end of this article.

This article is broken into two parts. First, we'll discuss the nature of texture maps and the issues involved in implementing a texture cache. Then, we'll look at some concrete implementations of caching systems used in games that are currently under development.



WULFRAM, the multiplayer tank from Bolt Action Software.

Textures and MIP-mapping

Texture storage is all about MIP-maps. MIP-maps are prefiltered versions of a texture map stored at varying resolutions. To simplify this discussion, we will focus on MIP-maps that are square and are a power-of-two in width (1×1, 2×2, 4×4,). We will speak of a MIP-map level (or MIP-level) as a nonnegative integer that describes the resolution of a MIP-map: a texture at MIP-map level n is 2^n texels square. MIP-level 0 is the smallest size at 1×1 texels, increasing with conceptually no upper bound (though we might voluntarily choose one to ease implementation) (Figure 1).

Jonathan Blow is vice president of software development at Bolt Action Software, a San Francisco-based game developer. He can be reached at jon@bolt-action.com

The reader should note that this MIP-level numbering convention is different from the most commonly used notation, in which MIP-level 0 is the texture at its maximum resolution, perhaps 128×128, level 1 is reduced by one step (that is, 64×64), and so on. That convention doesn't make any sense when you're deep into texture caching: what is the maximum resolution of a dynamically generated plasma fractal?

Let's take a look at the memory required to store textures. Every (uncompressed) MIP-map level of a texture requires four times as much RAM as the level below it. A texture at level n uses

$$Size(n) = \left(2^n\right)^2 = 2^{2n}$$

texels worth of RAM. Storing all MIP-map levels from 0 to n requires

$$SizePlus(n) = \sum_{i=0}^{n} Size(i)$$

Now suppose we have a texture at maximum detail and we want to store all MIP-maps down to level 0. How much extra memory does this require? In other words, how big is SizePlus(n) relative to Size(n)? We can figure this out using some standard power series diddling.

$$s = SizePlus(n-1) = \sum_{i=0}^{n-1} Size(i) = \sum_{i=0}^{n-1} 2^{2i}$$

$$2^2 s = 2^2 \sum_{i=0}^{n-1} 2^{2i} = \sum_{i=0}^{n-1} 2^{2(i+1)}$$

$$2^2 s = \sum_{j=1}^{n} 2^{2j} = \sum_{j=0}^{n-1} 2^{2j} + 2^{2n} - 1$$

$$2^2 s = s + 2^{2n} - 1$$

$$3s = 2^{2n} - 1$$

$$s = \left(2^{2n} - 1\right)/3$$

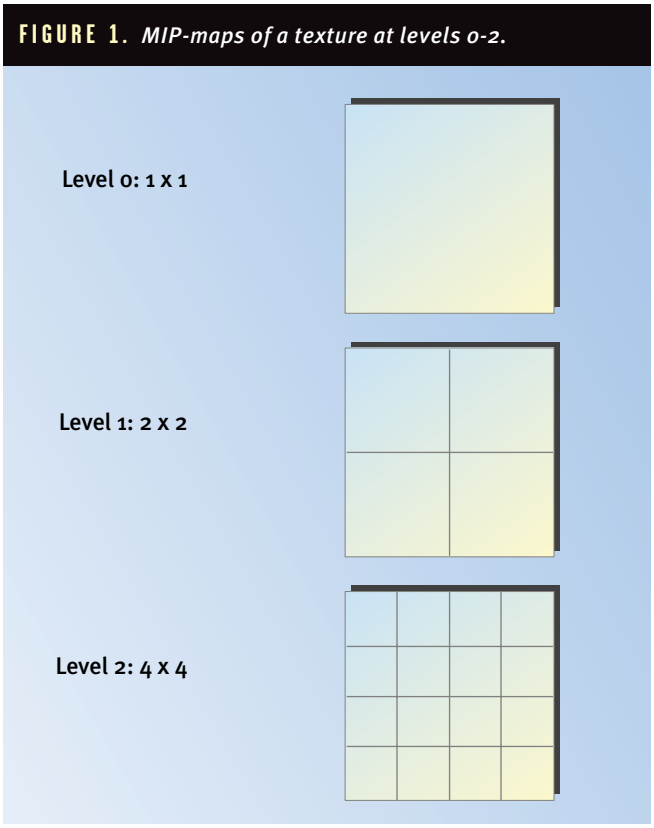
$$SizePlus(n-1) = \left(Size(n) - 1\right)/3$$

Since the amount of memory required to store a texture grows with 2^n as you climb the MIP-map ladder, you need to be careful about holding resolutions that are only as large as you really need. Conversely, because the required memory shrinks as you decrease MIP-map level, storing all the detail levels that are smaller than the level that you really need requires only one-third more memory.

Decisions Must Be Made

To build a system that's good at texture management, you want to keep only the necessary textures in RAM at only the necessary detail levels. The set of necessary textures will change from frame to frame based on what the 3D pipeline decides to do. To be effective, our caching system must predict and accommodate the needs of the pipeline. A good cache will be specifically designed for a particular application. (Does the cache need to handle dynamic or static textures or both? How many textures and at what sizes?) See Figure 2.

FIGURE 1. *MIP-maps of a texture at levels 0-2.*



Our system will have to perform the following tasks. There are a variety of ways to approach each problem, each with its own advantages and drawbacks.

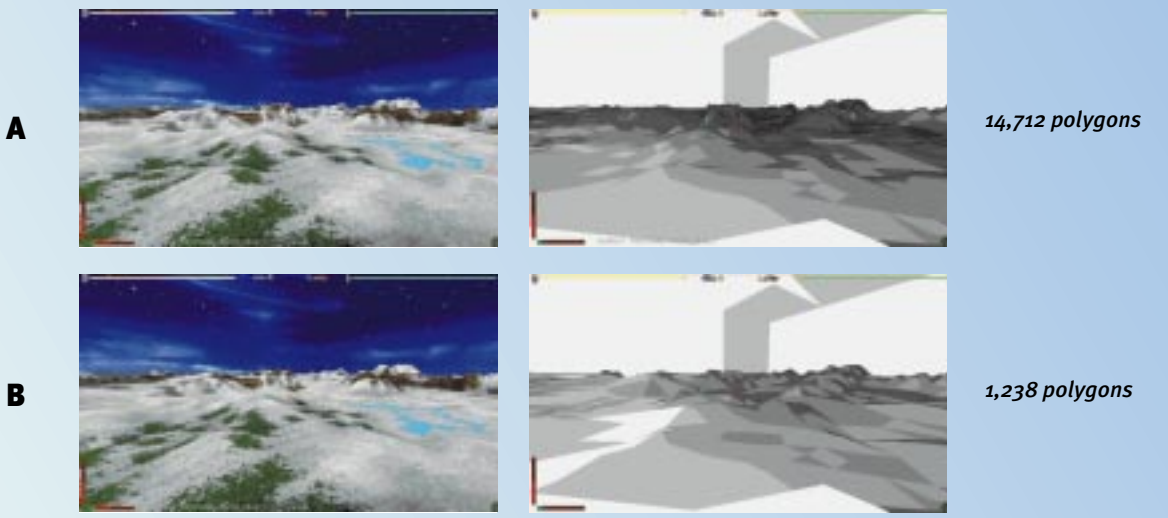
GENERATE TEXTURE REQUESTS. In order to fetch textures into RAM, the main 3D pipeline must tell the cache which textures it needs. Typically, this communication takes place in one of two ways. When the pipeline decides to emit a polygon using a particular texture, it calls a procedure to request that texture from the cache; this is known as pipeline hooking. Zoning, on the other hand, divides the world into zones; the pipeline predicts which textures it will need based on the position of the viewpoint, then requests those textures from the cache in batches.

Zoning generally requires preprocessing to determine visibility between spatial regions; however, it can be less expensive than pipeline hooking during run time because it does not require per-polygon tests. Pipeline hooking will generally require vigorous prefetching, but it is more versatile in that it places fewer restrictions on the application as a whole.

The concept of zoning applies itself more naturally to occluded indoor environments (such as QUAKE's) than to outdoor scenes (such as TERRA NOVA's).

DETERMINE THE DETAIL LEVEL. To operate efficiently, the cache should only retrieve detail levels that are required to draw the scene. We can achieve a precise solution by computing the texture gradients for each polygon that we emit and using those gradients, along with the nearest and furthest

FIGURE 2. *MIP-map level distribution for a typical scene from WULFRAM.*



On the left is the actual rendered image; on the right is a gray-coded version, with MIP-maps at level 0 drawn in dark gray, ranging up to level 7 drawn in white. Version A of the scene has detail reduction turned off; version B has it turned on. Note that A generally contains textures that are much smaller than B (the image on the right is darker). The point of this example is to highlight the interaction between cache design and pipeline design. The detail reduction system in WULFRAM strongly affects its texture usage statistics, changing the job that the cache must perform. A scheme that is memory-efficient, but behaves poorly with many small textures, would not suit applications with scenes resembling A, but would be fine for B.

vertices of the polygon, to tell us which MIP-map levels are needed. Alternatively, finding a conservative estimation involves cheaper computations than those required for the precise solution scheme. We can use a polygon's nearest vertex and a per-polygon precomputed coefficient to find a conservative upper bound on the necessary MIP-level. Or we can simply dodge the problem by always fetching textures at full resolution. We use the caching mechanism only to decide which textures are necessary.

Superficially, the conservative estimation scheme seems more attractive than attempting to find a precise solution because of our tendency to play bean counter with CPU cycles spent per polygon. However, since the conservative estimation scheme will generally request higher MIP-levels than the precise solution scheme, it places more load upon the cache. When comparing these methods in the WULFRAM engine, we found that on average the conservative estimation scheme would request textures one or two MIP-levels higher than would the precise solution scheme. This meant that the texture cache was fetching and synthesizing textures that were about eight times as large as they needed to be, slowing texture construction to an unacceptable rate. The frame rate also became very jumpy because texture-building costs are less evenly distributed than per-polygon costs.

However, the effect of the MIP-map level decision is application-dependant, and the conservative estimation scheme could be better than the precise solution scheme in some cases, especially if polygon counts are very high compared to the frequency of cache misses.

FILL THE CACHE. When the pipeline needs a texture, the cache makes it available. A synchronous fetching scheme momentarily pauses execution of the main program while the texture is being placed in the cache. Asynchronous fetching, on the other hand, allows execution of the main program to proceed in parallel with the cache filling process.

If textures are being dynamically generated into the cache in a CPU-bound manner, asynchronous fetching will only result in performance improvements on a multiprocessor machine. When a cache fill involves reading from external storage such as a bus-mastering hard drive controller, however, an asynchronous fetch won't actively consume CPU cycles.

EMPTY THE CACHE. The cache is of finite size. When it fills up, we must discard textures that are no longer needed. The games surveyed in the latter half of this article use a few different methods for emptying the cache. One common term we'll use is LRU, meaning Least Recently Used. In an LRU scheme, cache elements are marked with timestamps indicating the last time they were used. The element with the oldest timestamp is discarded.

MANAGE MEMORY. The texture caching system must find available memory for new textures efficiently. Also, if precious cache space is to be effectively utilized, fragmentation must be kept to a minimum. For a memory management scheme to be effective, it must be designed around permitted texture dimensions and the application's typical usage statistics. Developers are using a huge number of approaches to memory management; I've outlined several in the survey.

Potential Enhancements to Base Functionality

Once a caching system is in place, many things can be done to improve performance. Here are a few possibilities:

PREFETCHING. Fetching a texture invariably incurs a computational cost; requests for new textures will often occur in bursts, resulting in uneven demand on the CPU. To maintain the frame-rate level, fetch some textures before they're actually needed (during a lull in the handling of cache misses), thereby spreading each burst across more frames.

If texture fetching occurs asynchronously, it's possible that a texture won't have arrived in the cache by the time we need to draw it. In this case, we typically draw some sort of stand-in for that texture, which results in a loss of image quality. Prefetching textures, however, minimizes the possibility that our cache will be missing textures.

An easy way to prefetch textures is to bias MIP-level computations (recall the precise solution and conservative estimation schemes) so that larger MIP-maps are requested slightly early. In the very common case where the viewpoint is moving forward, this has the effect of automatically prefetching higher-resolution MIP-maps of visible textures.

COMPRESSION. If the cache system manipulates compressed textures, throughput requirements (and CPU requirements due to copying) will be reduced, and a cache of a given size will be able to hold more textures. However, this idea has many drawbacks. A software rendering system typically needs to manipulate uncompressed textures. Hardware accelerators use many different types of compression, so textures will often need to be uncompressed before they are sent to hardware (especially when using an abstracted 3D API such as OpenGL). The cost of decompressing your textures will usually outweigh the compression's initial benefits.

ADVANCED HIDDEN SURFACE ELIMINATION. Rendering scenes of high depth complexity using the painter's algorithm or a depth buffer places an unnecessary load on the texture cache; we will often load the cache with textures that are actually invisible because they're on polygons that are occluded by other polygons. A reasonable form of occlusion culling could reduce cache load tremendously for certain types of scenes.

Generating Lightmaps

Many upcoming games use "illumination maps" (or "lightmaps," the term used in the rest of this article). Lightmaps are a good example of dynamic texture generation: a source texture will often be combined with a lower-resolution lightmap to produce a shaded texture, which is then used on polygons in the scene. The shaded texture remains in the cache as long as it is being displayed. When a polygon's lighting changes, its texture is invalidated and recomputed using a new lightmap. The generation of the shaded texture can consume an appreciable number of CPU cycles, since effects such as bilinear filtering are often used to compensate for the lightmap's lower resolution (Figure 3).

In the survey of game engines in the latter half of this article, we'll speak of "lumel ratio." A lumel is one pixel of a lightmap, and the lumel ratio is a lumel's width divided by the width of a texel from the texture map to which the lightmap is being applied.

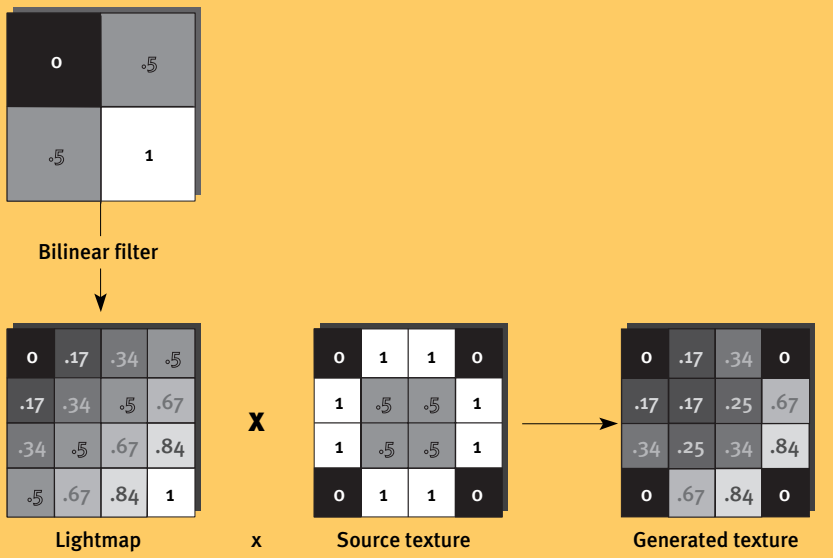


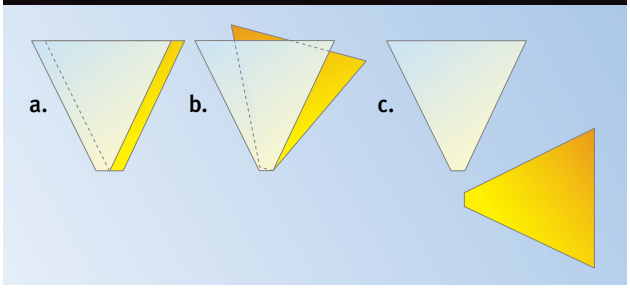
FIGURE 3. *A lightmap at level 1 is bilinear filtered up to level 2 so that it can be combined with a source texture at level 2 to generate the resultant texture. The lumel ratio in this example is 2:1.*

Performance Patterns

We would do well to note some basic truths about texture-caching systems. If the viewpoint and all objects in the scene are stationary, cache misses will be at a minimum because the scene will be the same from frame to frame. In this case, we only need to fetch new textures for animated polygons or when the cache is too small and we are forced to discard live textures. If we consider both these circumstances to be rare, then the cache is basically unstressed with a stationary viewpoint.

When the viewpoint moves linearly, small wedges of the frustum will come into view, and some textures that were already visible will shift to a higher MIP-map level (Figure 4a). This shift causes texture fetches, putting some load on the cache. The more quickly the viewpoint moves, the heavier the load on the cache, because more of those events are happening each frame. Backward motion places especially high stress on the cache because it introduces new textures at their highest detail levels (textures coming into view from the sides will generally be needed at intermediate detail levels).

FIGURE 4. *Regions of space revealed by a viewpoint moving in various ways: a is linear motion (sideways); b is rotational motion; and c is teleportation. The blue trapezoid represents the position of the view frustum before movement; the gold trapezoid is the frustum after movement.*



Rotating the viewpoint stresses the cache even more. At each frame, a wedge comes into view that widens with distance from the viewpoint (Figure 4b). Generally, the new volume of space revealed by a rotation will be much larger than the volume revealed through linear movement, resulting in a corresponding increase in cache events.

Long-distance "teleportation" of the viewpoint is the nastiest type of movement in terms of stressing the cache — the newly revealed area could potentially consist of the entire frustum (Figure 4c.)

Quality Loss

If cache filling occurs asynchronously and a texture isn't ready when we need it, we'll typically use a stand-in for the texture, such as a lower-resolution MIP-map. In this case, we suffer a loss in image quality. We can categorize potential magnitudes of quality loss in the same way that we just categorized cache stress: according to viewpoint movement.

A stationary scene should incur no quality loss in the steady state, again provided that no exceptional circumstances exist. Most linear motion in games is forward, and forward motion causes the fewest problems — when we're nearing a texture and it switches MIP-map levels from n to n+1, it does so at the point where there is almost no visual difference between level n and level n+1. This is the whole point of MIP-maps; if the texture arrives a frame or two late, the impact on the scene is minuscule.

Handling linear motion that causes new textures to come into view (say, sideward or backward motion) isn't too difficult because those newly-appearing textures are easy to prefetch. If a polygon comes into view at time t, then at time t-1 it was probably just outside the view frustum and was rejected during clip testing. This brings the polygon quickly to the attention of a properly concerned rendering engine.

Rotating the viewpoint can be slightly more difficult because the wedge widens as distance from the viewpoint increses. Nevertheless, in practice, a rotating viewpoint won't cause severe problems if adequate prefetching is in place.

Remember, however, that the quicker the motion, the greater the quality loss, because the area of newly-exposed polygons (which are now carrying erroneous textures) will be larger. Teleportation, therefore, is problematic because all textures will be unexpected.

These same principles regarding a moving viewpoint also apply to moving objects in a scene. Note, however, that rotating objects impose little stress on the cache. Textures on rotating objects first come into view edge-on, which means they are only necessary at minimal detail levels. As the polygon spins into view, the necessary MIP-map level rises in a continuous manner, just as it does for textures nearing a forward-moving viewpoint.

Survey of Engines in Progress

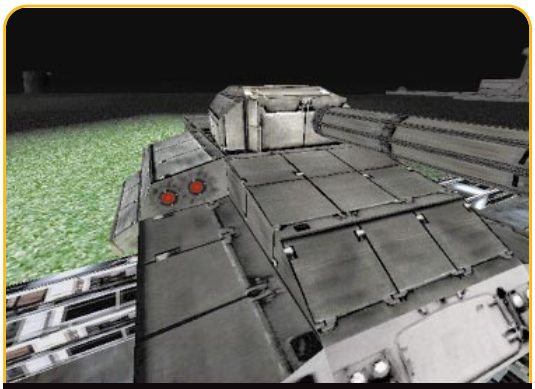
In the next section, we examine texture caching techniques in use by games and engines currently in development. Because these descriptions are only snapshots of prerelease software, they may not be accurately representative of the games in their final forms.

The main purpose of these descriptions is to provide examples of the design decisions that developers have employed to suit the games they are making. Because these game engines are very different from each other, it isn't useful to see these descriptions as "feature lists" in comparing engines to determine which is "better." All the developers involved have been very kind in sharing information about their systems and should be appropriately thanked.

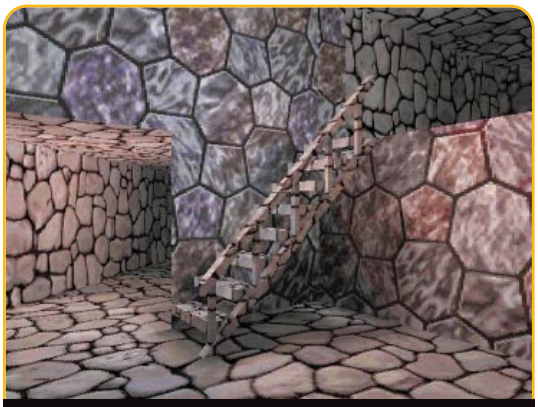
GOLGOTHA. According to Trey Harrison at Crack Dot Com, GOLGOTHA uses 16-bit textures that are initially JPEG-compressed; they are uncompressed at the beginning of each level, so the rendering engine only sees them as uncompressed. When unpacked, the textures are stored in the native formats of the current display (on 3Dfx Voodoo Graphics cards, for example, opaque textures are stored as 565 RGB, textures with holes are stored as 1555 ARGB, and textures with a full alpha channel are 4444 ARGB.)

Because of the high resolution and large number of textures in the game, textures reside primarily on disk and are cached into texture RAM. The game uses about 700 textures, typically 256×256 in size, requiring around 100MB of storage once they are uncompressed. Textures are power-of-two in width and height, but not necessarily square. The smallest handled MIP-map level is 16×16.

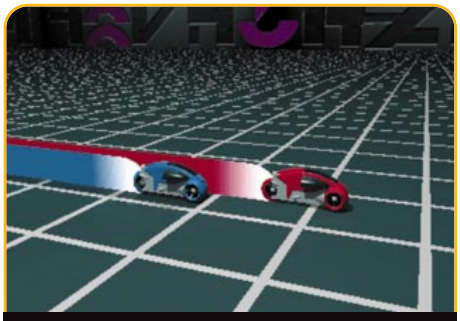
To determine which resolution of a texture it needs, GOLGOTHA uses a conservative estimation scheme based on the closest (1/z) of the polygons being displayed. When a new MIP-map is required, it's loaded asynchronously from disk (by a separate thread) into a



Golgotha, Crack Dot Com.



The Crystal Space engine by Jorrit Tyberghein.



The Hyper3D engine by John McCarthy.

temporary holding area in system RAM. At a fixed point in the rendering cycle, textures are downloaded from system RAM into texture RAM and removed from the holding area, eliminating concerns over the threadsafeness of 3D hardware APIs.

Until a texture arrives in memory, the MIP-map at the closest available resolution will be used for rendering. Some MIP-maps will always be available since the lowest resolution of each texture is always kept in texture RAM. Currently, the engine doesn't prefetch textures, although the final version of the game may implement this feature.

Texture memory is organized as a linked list of free space (used when finding memory for a new texture) and a linked list of allocated space (used when deciding which textures to discard from the cache). An LRU scheme is used to throw out textures when the cache is full.

Memory management becomes "a bit more ugly" under higher-level APIs such as OpenGL and Direct3D. With these APIs, the engine will discover that the cache is nearly full when an allocation request fails; textures must then be freed in a slightly blind manner because it's not possible to know what effect their deallocation will have on the fragmentation of the heap.

In order to keep the frame rate level, the main thread is limited in the number of textures that it can request per frame. The limit is imposed on both the number of textures and the total bytes requested and is adjustable based on system speed.

CRYSTAL SPACE. This is a freeware game engine under the GNU license. It was written by Jorrit Tyberghein. Crystal Space textures are 8 bits in depth and a power-of-two in both width and height, but not necessarily square. Most textures, however, tend to be square, typically 128×128 or 64×64. Four MIP-map levels are supported.

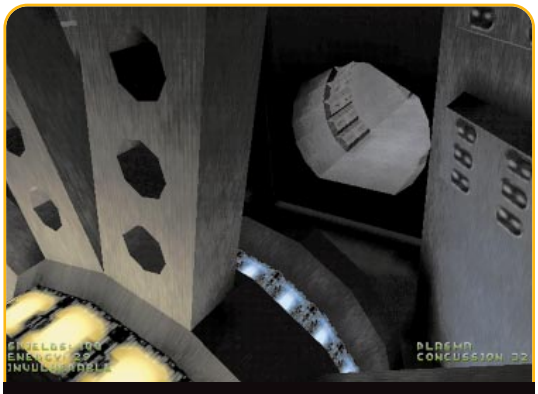
The engine uses lightmaps, which can be RGB or monochrome with a lumel width of 16:1 (see "Generating Lightmaps" for a definition of lumel). All source textures and static lightmaps are stored permanently in system RAM. The purpose of the texture cache is to manage the storage of textures generated by combining source textures with lightmaps. When a texture is built for a specific polygon, bilinear interpolation is used to expand the polygon's lightmap up to the necessary resolution; the source texture is tiled until it is the proper size. Since those 64×64 or 128×128 source textures can wrap across a polygon several times, the cached textures can be "very large." Since Crystal Space currently uses a software renderer, graphics hardware cannot impose a maximum limit on texture sizes.

The texture cache is of a fixed size; when it's full, an LRU scheme is used to discard textures until there is enough space. At present, C++ memory management (`new` and `delete`) is used for the texture cache, though this may be replaced with a custom memory manager in the near future.

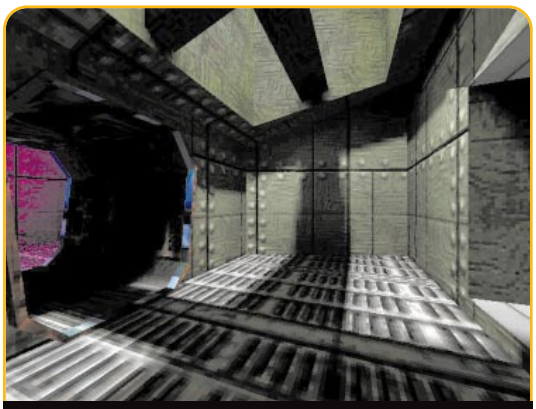
HYPER3D. This engine, written by John McCarthy, is made to render exterior scenes and space environments, with an emphasis on non-static objects. Textures are of arbitrary sizes and arbitrary multiple-of-8 bit depths. Each "color" of a texture is an 8-bit channel representing an arbitrary property; for example, a texture containing RGB color, alpha, and bump map information is stored in "40-bit color." Each channel is stored contiguously (a buffer consisting of two RGB texels would be stored as RRGGBB, rather than RGBRGB). The frame buffer is also divided into channels or "output banks." This memory organization simplifies the use of texture mapping operations to create complex effects. The banks of the frame buffer are combined into native format when it's time to display each frame. Because of this per-frame translation, the color model produced by mapping operations is not restricted to RGB; HSV textures can readily be used. Vertex-based lighting is supported, but because of the customizable screen mixing, lightmaps can also be used; they are treated just like any texture, then used to scale color in the final buffer synthesis.

Because of the high resolution and depth of textures, they are stored on disk and demand-loaded into system RAM when the renderer tries to use them. Loading occurs synchronously at a fixed point in the update cycle; the number of texture loads per frame is capped at a maximum (adjustable) value to keep frame rate level.

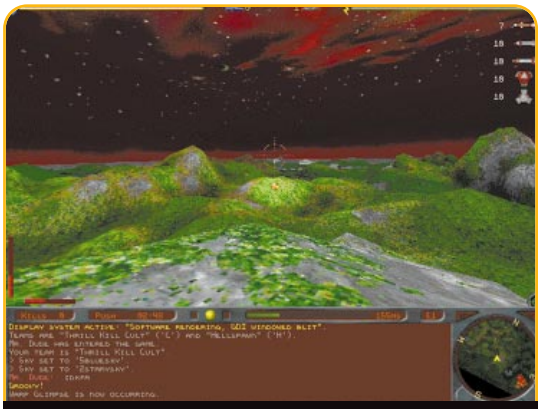
When textures need to be displayed but are not yet loaded, the renderer draws a flat-shaded polygon instead. As Hyper3D author John McCarthy says, "Frame rate was more important than perfectly correct scenes."



DESCENT 3 from Outrage Entertainment.



KAGE, Terminal Reality Inc.



Wulfram, Bolt Action Software.

The texture cache is a linked list of 256×256-byte blocks. The cache manager fits textures together within these blocks. To reduce fragmentation, textures are rounded up to the nearest 8-texel boundary in x and y (so a 73×131 texture will take the same amount of space as a 79×133 texture: they will both occupy an 80×136 aperture). When a new texture won't fit into an available block, textures are discarded based on their reference counts (textures being used 0 times are thrown out first, then textures being used 1 time, and so on). If the heap becomes too full or very fragmented, currently-used textures will have to be discarded and reloaded during the next frame. These discards tend to reduce fragmentation.

DESCENT 3. According to Jason Leighton at Outrage Entertainment, DESCENT 3's textures are stored in 16-bit color; they are all square and power-of-two in width. QUAKE-style lightmaps are used, using lumels at a size of 16:1. Because there are many lightmaps and each lightmap is small, the caching system must efficiently handle large numbers of small textures, in addition to being effective for larger textures.

The main purpose of DESCENT 3's caching mechanism is to efficiently use the texture RAM of a 3D accelerator. All textures from the current game level are held in system RAM, with the cache pulling them into texture RAM as needed.

The square, power-of-two texture geometry allows the system to manage texture RAM without the possibility of fragmentation. Texture memory is divided into an ordered series of blocks, each of which stores textures of a given MIP-map level. There are seven such regions, with the smallest storing 2×2 textures and the largest storing 128×128 textures. Suppose that the renderer needs to use a new 16×16 MIP-map, but the 16×16 region is full. First, the system checks the 16×16 region to see if any textures can be discarded. If so, the new texture is simply uploaded over the old one. If not, the 16×16 cache region needs to grow. It can grow to the left, taking space away from the 8×8 region, or to the right, stealing space from the 32×32 region (Figure 5). Heuristics are used to make the right decision about which way to grow.

Note that when a region grows to the right, it steals only one texture from the neighboring region and gets enough space to hold four of its own textures. When growing to the left, a region must consume four of its neighbor's textures to produce space for only one of its own. When memory is stolen from a region, any textures residing within the reassigned memory are discarded.

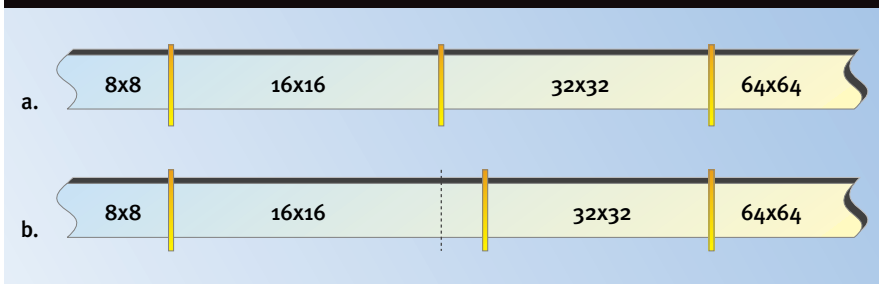
This memory management scheme requires direct control over a single, continuous memory aperture. Many hardware-specific APIs (such as 3Dfx's Glide) support this type of access naturally. However, when using higher-level APIs such as OpenGL or Direct3D, such direct control is lost, and this sort of scheme becomes difficult or impossible.

KAGE. Terminal Reality Inc. designed this an engine to render arbitrary polygonal scenes generated in editors such as 3D Studio MAX. According to Paul Nettle, KAGE uses lightmaps with a lumel width of 4:1. It uses a precise solution scheme to determine the MIP-map levels of polygons in the scene.

Textures are of unconstrained size, but their width or height cannot exceed 256 texels. The cache is stored as an array of pages. Each page is a two-dimensional image and is indexed by a list of occupied space. Within a page, textures are lined up from left to right.

Each page stores textures only within a certain range of texture heights. The global page array can be constructed so that each page height is a multiple of n. (If n is 8, there will be pages that are 8 texels high, pages that are 16 texels high, and so on. Since the height is capped at 256, there would be 32 different page heights.) Textures are effectively rounded up to the nearest multiple of n in height and stored in the appropriate page; the gap between the bottom of an allocated texture and the

FIGURE 5. *DESCENT 3's memory management scheme. a: Memory is partitioned into contiguous regions based on texture size. b: The 16x16 region has grown by taking space from the 32x32 region.*



bottom of a page is wasted memory. In practice, the amount of waste produced is very small.

When it is time to allocate space for a surface, the system searches pages of the appropriate size for a "best fit." This search is capable of examining continuous groups of allocated areas and gaps between them to determine whether that area should be cleared for the new texture (by comparing the combined most-recently-used values of allocated shards and taking the lowest).

WULFRAM. In this game from Bolt Action Software, textures are 8 bits deep, square, and power-of-two in width. At any given time, the game will have random access to around 2,800 textures, about 1,500 of which are generated at the beginning of each level. Most textures are 128×128, and all MIP-map levels of every texture are stored on disk — the size of the texture store is around 58MB. The caching system's main emphasis is on caching textures from disk in system RAM.

Detail reduction is employed heavily in landscape rendering using an algorithm based on Peter Lindstrom's SIGGRAPH paper (see References). To accommodate this detail reduction, the engine dynamically generates textures that are composites of the source textures in the texture store. Monochrome lightmaps are also used; lumel width varies within the scene, but the average is around 16:1.

A dependency management scheme is employed to synchronize the fetching of multiple textures from disk to build a composite. MIP-maps that are members of visible composites are locked in system RAM so that they are readily available when the composite needs to be resized or broken into smaller pieces.

Prefetching is used in several places. Animations (such as explosion bitmaps) are fetched several frames ahead, and the first few frames of each animation are always locked in system RAM at their maximum detail levels. Sky textures near the edges of the view frustum, as well as ground textures very close to the viewpoint, are prefetched.

MIP-map levels 0-3 of all textures are kept in system RAM at all times. The fetching of textures from disk is performed asynchronously, and whenever a texture is not yet ready, its highest available MIP-maps are used.

The texture cache is not limited in size, but it's generally kept small by a cache sweeper, which looks at 1/16 of the textures in the cache at each frame and discards those that have not been used for 300 milliseconds. After some reflection, this appears to be a less than optimal cache design, but it's adequate for present purposes. The texture cache generally occupies between 2MB and 4MB of system RAM when resolution is set to 640×480. ■

Acknowledgements

Many thanks to Trey Harrison at Crack Dot Com, Crystal Space author Jorrit Tyberghein, John McCarthy, Jason Leighton at Outrage Entertainment, and Paul Nettle at Terminal Reality Inc. for their indispensable help in supplying information for this article.

REFERENCES

Caching graphics in a game is certainly not a new practice. See Jonathan Clark, "Object Cache Management," *Game Developer*, February/March 1996, for a discussion of data caching in the 2D game ABUSE; this article will serve as a good introduction for those not used to thinking about caching.

For implementations of simpler caching schemes used by successful games, the reader is referred to the source code to ABUSE (www.crack.com/games/abuse) and DOOM (ftp://ftp.idsoftware.com/idstuff/source).

More information about KAGE can be found at www.terminalreality.com/engine/kage.html. For other technical info and some research papers that were influential in KAGE's design, see www.grafix3d.dyn.ml.org.

Crystal Space executables and source code can be found at www.geocities.com/SiliconValley/Horizon/3856.

Information about GOLGOTHA, including demo executables, can be found at www.crack.com/games/golgotha.

Technical descriptions of WULFRAM's implementation can be found at www.bolt-action.com.

General information about DESCENT 3 can be found at Outrage Entertainment's web site, www.outrage.com. At present, there is not much technical information, but the site promises to include developer comments in the future.

John McCarthy's home page, including some tidbits related to Hyper3D, can be found at www.geocities.com/SiliconValley/Peaks/6846. He can be contacted at John@McCarthy.net.

A good explanation of lightmaps can be found in Michael Abrash's "Quake's Lighting Model: Surface Caching," *Dr. Dobb's Sourcebook* #260, November/December 1996.

Zen of Graphics Programming (Second Edition) (The Coriolis Group, 1996) by Michael Abrash is considered a classic.

You can get details about computing texture gradients in screen space in Chris Hecker's "Perspective Texture Mapping Part 1: Foundations," *Game Developer*, April/May 1995.

Hin Jang's "Tri-Linear MIP Mapping," available at www.scs.ryerson.ca/~h2jang/gfx_c.html, contains a good introduction to MIP-mapping.

Peter Lindstrom's "Real-Time, Continuous Level of Detail Rendering of Height Fields" (*SIGGRAPH 96 Conference Proceedings*) outlines an algorithm that's handy for landscape rendering.

Paul Nettle's "The KAGE Surface Caching Mechanism" is available at www.terminalreality.com/engine/kage.html

Discrete Mathematics and its Applications (McGraw-Hill, 1991) by Kenneth H. Rosen is a good text for helping you figure out power series.

Functions used to determine texture detail levels can be found in section 3.8.1 of *The OpenGL Graphics System: A Specification (version 1.1)* by Mark Segal and Kurt Akeley. It's available at www.sgi.com/Technology/OpenGL/glspec1.1/glspec.html.

Choosing which elements of a cache to discard is a well-studied subject. For starters, look up LRU in *Operating System Concepts* (Third Edition) (Addison-Wesley, 1991) by A. Silberschatz, J. Peterson, and P. Galvin.

A Fa a Yo Wa o Be: I el VT e 2.5

by Ron Fosner

Programming today's PCs is much more complicated than in years past. Multitasking operating systems, multithreaded programs, multiprocessor systems, multipiped CPUs, and multicached CPUs contribute to programming headaches. All of these new layers have to be taken into account if you want to assess how your program spends its time. Still, many of these aspects are beyond the comprehension of all but the most prickly programmer. Fortunately, there are a few products on the market that make this task easier; one such product is Intel's VTune.

When Intel designed the Pentium Pro class of processors, it built in instrumentation that allows you to identify problems such as cache misses. VTune uses these counters to profile your code, which makes it fairly simple to gather a host of information about your program's execution.

Why Profile?

Here's an example of an instance in which profiling comes in handy. Recently, one of my clients wanted me to use OpenGL to render a scene for use as a texture. Everything was fine until they wanted to alpha-blend the texture. "Ummm... software renderers typically don't support destination alpha," I mumbled. After doing a quick check of pixel formats using both Microsoft's and SGI's OpenGL .DLLs, I confirmed that destination alpha wasn't supported in any mode. (Destination alpha is a display buffer that supports the A in RGBA, not just the usual RGB of the typical raster display. In OpenGL software renderers, the alpha is used up to the last color calculation, then is lost in the write to the screen buffer.)

Essentially, my problem was that I had a buffer filled with RGBRGBRGB... byte values that I needed to convert to RGBARGBARGBA... format. My first step was to create a quick program to copy from one buffer, smash in an alpha value, and then write out to another buffer. The code looked like the following:

```
void Simple( int w, int h, char* src,
     char*dest )
{
  int i,j;
  for ( i = 0 ; i < w ; i++ )
    for ( j = 0 ; j < h; j++ )
      {
      // take an RGB, make it an RGBA
      dest[0] = src[0];
      dest[1] = src[1];
      dest[2] = src[2];
      dest[3] = 255;  // max alpha
      src += 3;       // src is RGB
      dest += 4;      // dest is RGBA
      }
}
```

Ron Fosner works on fast 3D rendering code at Data Visualization, is the author of OpenGL Programming for Windows 95 and Windows NT *from Addision-Wesley, and is teaching a course on code tuning at the 1998 Computer Game Developers Conference, so stop in and say "Hi" if you're in town.*

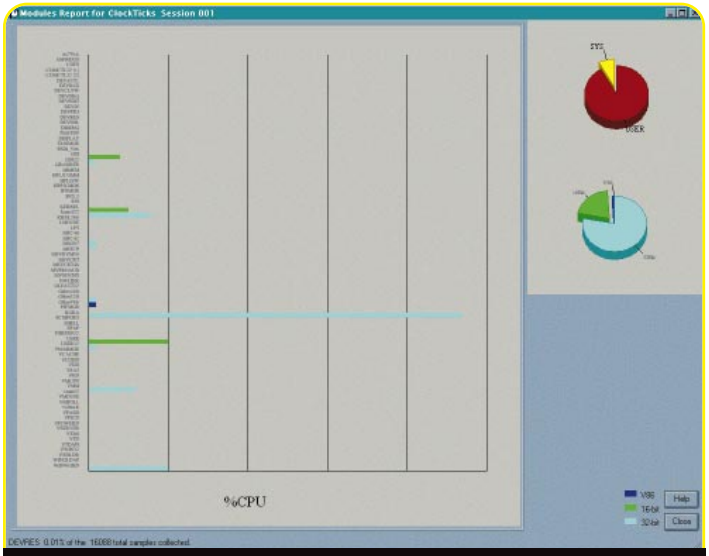


FIGURE 1. *Opening a profiling session in VTune.*

FIGURE 2. *VTune's hotspot window.*

This was a simple, hard-to-screw-up implementation. The values came into the loop byte-by-byte and I simply slapped on an alpha value of 255 to make the texture totally opaque. (The client would do some processing later.) Now, being the savvy coder that I am and knowing in advance that I had to write this review, I proceeded to write five more versions of this function. First, I switched over to using pointers, then I wrote a version that did an in-place, reverse **memcopy**. (Assuming that we're given the memory for an RGBA bitmap, we read the RGB values into this space and then start grabbing values off of the end of the RGB area and filling the RGBA area, eventually meeting at the beginning of the bitmap.) In addition to bytes, I did another version that operated on **word**s and finally, one that processed 32 bytes at a time. All of these functions were in a test harness so that I could easily validate the results and play with the functions. This, then, would be my project with which to test VTune's capabilities.

On to The Profiler

Installing VTune was no problem. The basic installation took up about 45MB and went flawlessly. Setting up VTune was easy. A wizard took me through the steps of creating a profiling project and associating it with code. My first problem arose when I ran out of disk space for the database files and a dialog box popped up with the message "Assert in barn.c line 666. 1 == 0." Now, when I see this kind of thing in production code, it doesn't exactly fill me with confidence. It took me little time to figure out what was causing this particular problem. I cleared some disk space, tried again, and this time everything worked. After watching three different progress meters complete their jobs, I was asked if I wanted to "Open the session." Clicking O.K., I was presented with the fairly cluttered graph seen in Figure 1.

This figure illustrates my biggest complaint against VTune. It uses multiple windows to display information. To get successively more detailed information about your program, you drill

Terminology for the Pentium Family of Processors

U and V pipes. The Pentium family of processors has two execution pipelines that can operate in parallel. If you have a smart compiler or know assembly, you can theoretically fill both pipelines and execute two instructions per cycle. Each pipeline consists of five (for Pentium) stages, in which five instructions can be overlapped. Instructions have to fall into the prefetch, decode, execution, write-back order. (The Pentium Pro and Pentium II have a twelve-staged architecture.) What you should understand is that the order of instructions has a great effect upon execution speed. In addition, the V pipe can only process certain instructions.

AGI. Address Generation Interlocks occur when you calculate an address and then use it immediately. Typically, AGI causes a two-clock stall. However, this only occurs on the Pentium, and AGI offers a big performance gain if you're running on a Pentium Pro or Pentium II.

L1/L2 Cache. The L1 cache is an 8K or 16K on-chip buffer (one for data and one for code) in which data can be fetched in one cycle. The L2 is the secondary cache (off-chip on the Pentium) and it's typically 256K to 1MB. If you can keep loops within the L1 cache size, your program execute much more quickly. If you can prefetch data before you need it, you won't have to wait for the data to become available.

BTB. The Branch Target Buffer contains a history of mispredicted branches. The processor actually tries to predict which way branches will go and maintains a buffer of mispredicted branches. Keep loops small, since the buffer holds only 256 addresses. Large loops with lots of braches can swamp the buffer, thus slowing down your program.

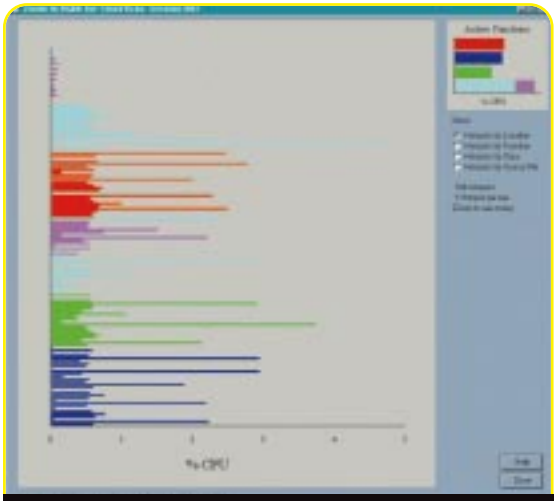


FIGURE 3. *Hotspots location-based display.*

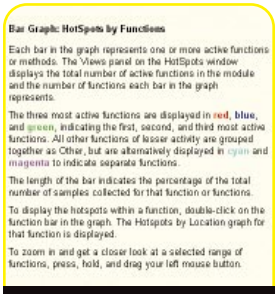


FIGURE 4. *Pop-up help example.*

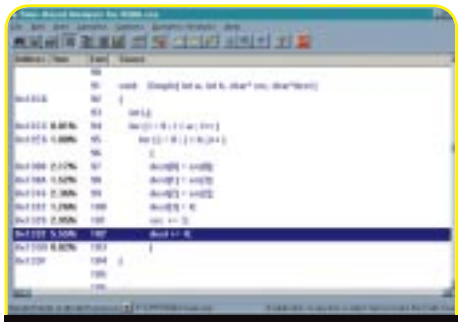


FIGURE 5. *VTune's source-code display.*

down (read, double-click), which pops up another window on top of the last one. It wasn't unusual for me to have six, eight, ten, or more windows cluttering my screen when using VTune. And since I was usually only interested in the information in the most recent window, I had a bunch of useless windows sitting on my desktop.

Notice that the screen in Figure 1 is pretty cluttered for a program that only has eight functions. That's because the first window that VTune brings up is of system-wide activity listed in alphabetical order. Using the minuscule lines of that you can see in Figure 1, I could eventually find my program and double-click to find out more information. Because I was careful about how I set things up, the long cyan line in Figure 1 is my program. Double-clicking on that line brought up another window containing the program's hotspots. With this window, you can select hotspots by function, location, source file, or, if you're profiling a Java application, by Java class (the Java-class option is nice, but I'd like to be able to view by C++ class as well). A hotspot window is shown in Figure 2.

The hotspot window is probably where you'll start the initial interpretation of your profiling results, since it's the most useful. While VTune uses many annoying windows with tiny lines, its user interface also has some bright spots. In the hotspots window, you can not only select source file, location, or function, but if you hover your cursor over a line, a pop-up information box displays that line's properties (in this case, percentage of CPU and actual number of samples). In addition, you can right-click and drag the cursor to zoom in on a particular area of the display. Or you can simply double-click on a single line to look at a particular source file, function, or location. This navigation method lets you easily drill down into a particular file or part of a file, but you can get carried away and really clutter up your screen this way. For large programs with thousands of functions, you'll have to do some judicious clicking on some very small lines to get any meaningful information out of the data. Figure 2 shows the window after I had selected the various swabbing functions. The small cyan bar on top is the main function, and the other bars are the various versions of the swabbing function. The bars are color-coded, with red, blue, and green indicating the three most CPU-intensive functions. The remaining functions are alternately displayed in cyan and magenta. Switching to a location-based display brings up Figure 3.

In addition to having pop-up information boxes all over the place, you can right-click on a window and get a pop-up help menu. The one for the hotspots window is shown in Figure 4. These menus are very effective in helping you figure out the interface and some of its hidden power.

In Figure 3, you see a long cyan line. This single line displays a hotspot in my initial code. If you double-click on that line, you enter a source code–based display. In order to get VTune to work with your source code, you'll have to provide it with some method of determining how its sample database relates to the source code. This is where I ran into my first serious problem with VTune. In my initial attempts to get my source code displayed, VTune repeatedly told me that no source was available. Yet I had built a debug version and had correctly identified the location of the source. Delving into the Read Me file and the well-crafted online help, I discovered that VTune requires a Visual C++ .DLL in order to decode a .PDB file. No problem, that file was present. The help file mentioned that this .DLL was probably located in the x:\msdev\bin directory and in any event would have to be located somewhere along the path. Now, with Visual C++ 5, Microsoft has started sharing .DLLs, and it stores these DLLs in a SharedIDE\bin subdirectory. Adding this directory to my path solved VTune's problem in locating the source.

I noticed that Intel's web site hosts a VTune discussion. There were lots of appeals from fellow programmers crying that they could not get VTune to display source with their Visual C++ programs. I noticed that there were no replies from any Intel folks as to how to solve this problem. Hello Intel!

Once that problem was solved, I provided a preprocessed source file (using the /P option on the compiler line — you can also provide a makefile so that VTune can preprocess your source code), and soon my VTune displayed my source, along with timing information. From Figure 2, double-clicking on the top magenta line (the line for the simple swabbing function) and then double-clicking on the longest line in the display (which corresponds to the longest line in Figure 3), gives us the source code display, shown in Figure 5.

This is where you have to be careful. With instruction pairing (remember those U and V pipes), the actual percentages shown are typically smeared across instructions. In assembly view, for example, the times associated with a particular instruction are typically off

by one instruction, sometimes by two because of pairing issues. So some judicious interpretation of timing results is in order. Most programmers will be content to stop at this point — you have a time associated with functions and source lines. However, if you ever need to delve into a more detailed view or if you frequently need to code in assembly, then VTune is more than capable of going the distance.

If you display the mixed assembler and source, you'll get the display shown in Figure 6. Here, each assembly line is displayed with a time. In addition, there's a lot of other information on the screen. The red and blue blocks on the right side of the window indicate pairing issues with the U and V pipes. There are also blocks that indicate pairing problems. With some reordering of commands, it's possible to alleviate some of the pairing issues, but in practice I've found that there's only so much you can do in C or C++. If you're programming in assembly, you have many more opportunities for optimizations.

On the right side of the window are listed problems associated with the particular commands. For example, notes that start with `Exp_AGI` are referring to commands that directly require information from the previous operation and result in an Address Generation Interlock. For example, fetching a value and then using it immediately will typically result in an AGI. Reordering the operations, such as sticking an unrelated operation from another part of the loop in-between the fetch and usage, will alleviate the AGI condition. VTune is excellent at pointing out these types of issues, and with a little bit of practice, you can train yourself to avoid coding them. VTune includes a Code Coach, which will make suggestions as to how you can modify your code to make it faster. The Code Coach currently works only with C and ASM, but VTune 3.0 will include C++ support.

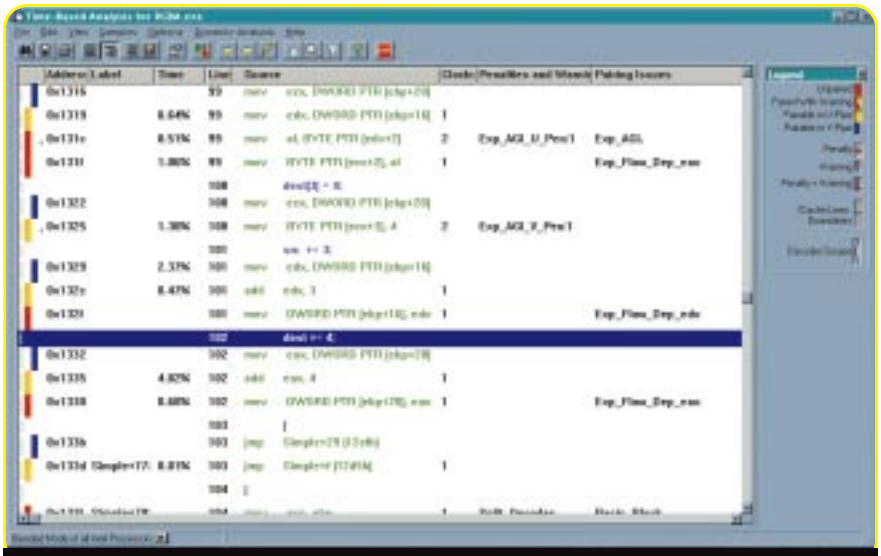


FIGURE 6. *Where VTune shines — the disassembly display.*

VTune 2.5

RATING (OUT OF FIVE STARS): ✪✪✪✪

Intel Corp.
Santa Clara, CA
408-765-8080
http://developer.intel.com

Price: $199

Software Requirements: Microsoft Windows 95 or NT 3.51 or higher.

Hardware Requirements: '486 (Pentium or better recommended), 32MB of RAM, 40MB of disk space, CD-ROM.

Technical Support: Web-based and hotline support

Pros:
1. Easy to install and use.
2. Excellent processor support.
3. Gives you a lot of information for the price.

Cons:
1. User interface needs some work.
2. Designed only for Intel processors.
3. Might be too much information for those who don't care about processor issues or ASM programming.

In short: For the price you get a lot of stuff. If you just need a basic profiler and you can find one cheaper, then go for it. But for $199, I certainly can't complain about all the "extra" things. Version 3.0 will sport a much-improved user interface, but at the higher price of $279. This new price is getting close to the point where it's tough to get an automatic "OK" from your boss. Still, if it helps boost your program's speed by 10%, it's still a bargain.

Is It Worth It?

You might wonder if purchasing a profiler is really worth the expense. In fact, you'll really never know until you profile. If you examine Figure 2, you can see why. I wrote six separate versions of code to do essentially the same thing. In Figure 2, the total time spent in the program is displayed from the top with main, simple array access, pointer access, quad access version 1, quad access version 2, word access, and byte access. I expected the pointer version to be slightly faster than the one using array access, the version that accessed 16 bits to be faster than the pointer version, and finally the 32-bit memory access to be faster than the 16. It turns out that the simple, array access function that I started with (the top magenta line) is faster than both the pointer version and the byte access version, finally tying with the 16-bit access version — and the simple version uses two buffers! In fact, it turns out that doing the complicated math required for the

32-bit version was the only thing that made it faster. Apparently, my initial assumptions about how the code would profile were almost totally wrong. This isn't an unexpectd result of profiling, and I wasn't dissapointed by it. My experience illustrates the fact that you often just don't know how fast something is until you measure it. So if you're still guessing as to where the slow areas of your program are, do yourself a favor and try measuring it. I guarantee, you'll be quite surprised.

The basic VTune setup uses sampling (halting the processor every few cycles and seeing where the code is executed) to get a profile, but if you have a Pentium Pro or Pentium II processor, you can perform event-based sampling (EBS), which measures performance related processor events such as data cache misses. These types of events are important if you process a lot of data — such as sending a lot of vertices across the bus or dynamically altering texture maps. In addition to the basic program profiling capabilities that VTune comes with, it also has a static code analysis function that will work on .EXE, .OBJ, .VxD, or .DLL files. If source code is available, VTune will use it; otherwise, it will generate assembly. The static code analysis looks at the code, estimating the expected performance and pointing out performance changes you can make. Another option is to use the Dynamic Analyzer to dynamically analyze and fine-tune a small section of your application. The Dynamic Analyzer executes the application, traces its execution, then simulates and monitors any portion of code that you specify. This is a way of profiling only a small section of code for a very detailed analysis. These options generally take longer than just plain sampling because VTune is doing a lot more work.

Besides VTune, the CD contains the Intel C/C++ and Fortran compilers. You can also install a plug-in for Microsoft Visual C++ that will allow you to use the Intel C/C++ compiler in the Developer Studio. The Intel Performance Library Suite includes a signal- and image-processing library and a math kernel library, which have been optimized for MMX processing. Also included is some Intel architecture documentation, which contains information on writing fast Pentium, Pentium Pro, and MMX code.

In addition to the regular features that you'd expect from a profiler, VTune is capable of monitoring the entire system. It is designed to assist in finely tuning code, especially at the assembly level. It's surprisingly easy to get up and running with VTune, and on the whole the online manual is thoughtfully laid out and the hypertext links are quite useful. On the negative side, the user interface quickly clutters up your desktop, and the focus on assembly might be a bit overwhelming for those who don't care to get such a detailed report at such a low level. However, if you or someone on your team isn't scared of assembly, then VTune can be a valuable tool for discovering what is actually going on with your program. The online documentation (essentially, the only documentation) was fairly complete. Intel provides both web-based support and a hotline phone number. When I ran into a problem identifying my source location, I put in a call to Intel's hotline and left a message on an engineer's voice mail. I received a call back within 24 hours, and after playing some phone tag, I found the support call only moderately helpful. But support did call back to make sure everything was O.K.

By the time you read this, Intel's VTune 3.0 should be almost out. Its features are supposed to include a single screen UI and an improved C++ and ASM coach, plus better support for Pentium II processors. Stay tuned to see if the support improves. ■

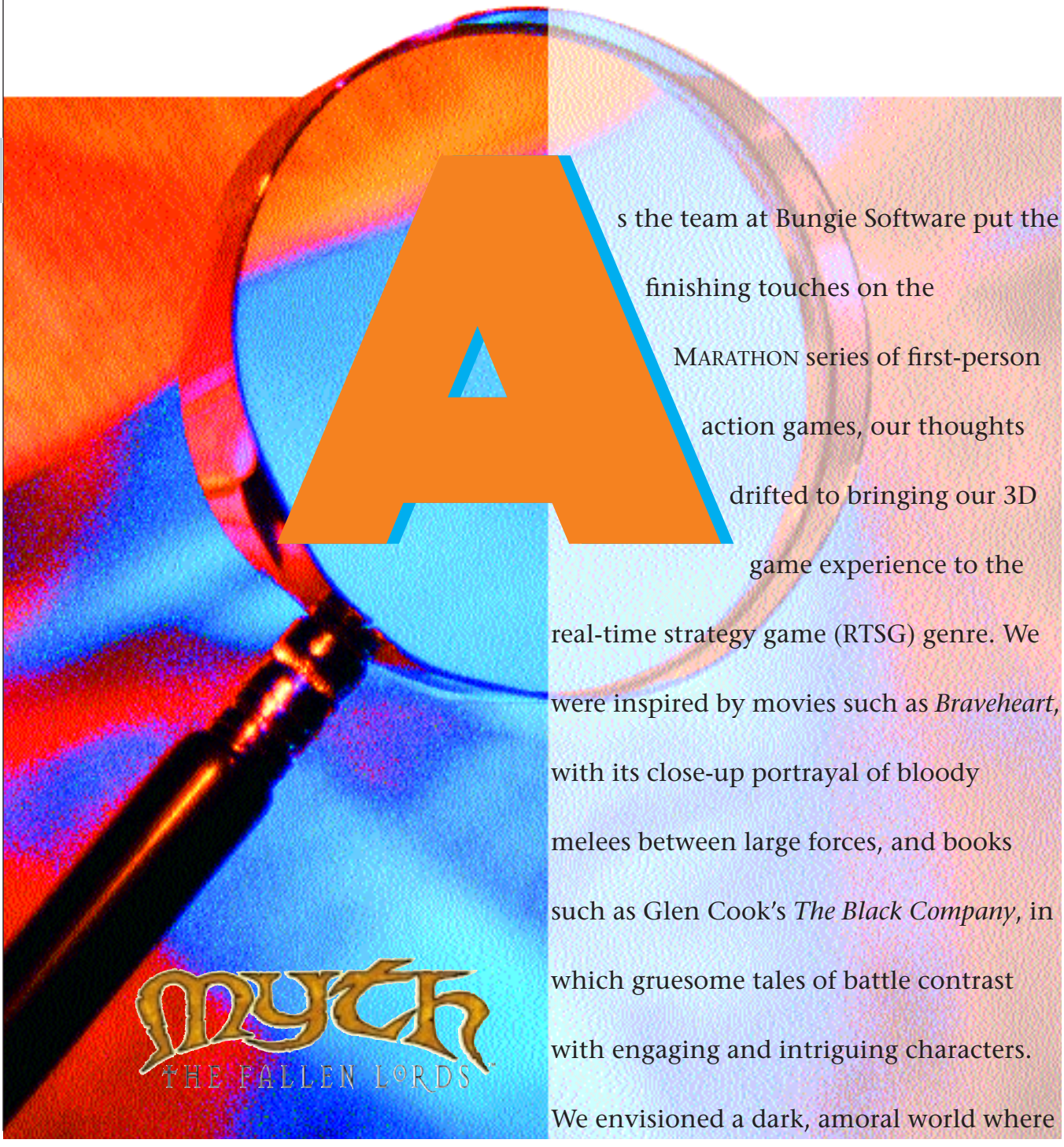
FOR FURTHER INFO

If you're looking for good books on tuning programs, I have to recommend two. *DirectX, RDX, RSX, and MMX Technology* by Rohan Coelho and Mike Hawash ($44.95 from Addison-Wesley, 1997) is highly recommended for all levels of high-performance programmers. It includes information on DirectX 3 and a VTune tutorial (the authors work for Intel). The CD includes a trial version of VTune.

For those of you who live and breath assembly, try *Inner Loops* by Rick Booth ($39.95 from Addison-Wesley, 1997). This book gives in-depth explanations about how to write fast inner loops and gives many examples that show how changing one line of code can effect a program's speed, sometimes by 25%.

MYTH: THE FALLEN LORDS

by Jason Regier



As the team at Bungie Software put the finishing touches on the MARATHON series of first-person action games, our thoughts drifted to bringing our 3D game experience to the real-time strategy game (RTSG) genre. We were inspired by movies such as *Braveheart*, with its close-up portrayal of bloody melees between large forces, and books such as Glen Cook's *The Black Company*, in which gruesome tales of battle contrast with engaging and intriguing characters. We envisioned a dark, amoral world where

opposing sides are equally brutal and their unity is torn by power struggles within the ranks. We dreamed of game play that combined the realism and excitement of action games with the cunning and planning required by strategy games.

Our original design document, if you could call it that, was simply opposing lists of "Stuff that Rocks" and "Stuff that Sucks." Anything vaguely cliché, such as excessive references to Tolkien novels, Arthurian legend, or "little boys coming of age and saving the world," went in the "Sucks" category. The "Stuff that Rocks" list was filled with ideas that contributed to the visual realism of the game: a true 3D landscape, polygonal buildings, reflecting water, particle-based weather, "blood-spattered battlefields littered with limbs," explosions that send shock waves through the terrain, and "lightning frying guys and their friends."

Our goals for the product were lofty: simultaneous release on Windows 95 and Macintosh platforms, integrated Internet play, and a free online service to allow players from across the globe to battle one another. From this vision, MYTH: THE FALLEN LORDS was born.

The MYTH development team. From left to right: Mark Bernal (artist), Frank Pusateri (artist), Rob McLees (artist, holding statue of a Trow), Jason Regier (programmer), Jason Jones (programmer/project leader) Marcus Lehto (artist). Not pictured: Ryan Martell (programmer), Tuncer Deniz (product manager), Jay Barry (level design).

The Making of a Legend, er, Myth

The project began in January 1996 with four programmers, two artists, and a product manager; midway through development, one programmer dropped out and an artist was added. Music, sound effects, and cut scenes were done out-of-house, and a few artists were contracted to help with interface artwork.

The roots of the MYTH programming team were on the Macintosh, so most initial coding was done on the Mac with Metrowerks CodeWarrior. When PC builds were required, though, we used Microsoft Visual C/C++. MYTH was written entirely in C.

In addition to creating the shipping product, we developed four tools to aid in the construction of the game. One utility, the Extractor, handled the importing of sprites and the sequencing of their animations. Another tool, dubbed Fear, dealt with importing polygonal models such as houses, pillars, and walls. The Tag Editor was responsible for editing the constants stored in cross-platform data files, which we called tags. And finally, Loathing, our map editor, handled the rest. Loathing was built around the MYTH engine and allowed us to modify the landscape, apply lighting, set terrain types, script the AI, and place structures, scenery, and monsters.

The artists used Alias|Wavefront's PowerAnimator and StudioPaint on a single Silicon Graphics Indigo 2 to create polygonal models and render all the characters. At one point, the artists worked separate day and night shifts so that they could maximize their time on the SGI. Models were brought into the game using Fear, while the sprites were cleaned up in Adobe Photoshop and imported with the Extractor. To create the texture maps for the terrain, the artists used Photoshop to draw what looked like an aerial photo and applied it to a 3D landscape in Loathing.

If this sounds like a lot of work to you, you're right. Most maps took at least a week or two to create. We considered using fractal-generated landscapes, but we were worried that the inherent randomness of such terrain would make it extremely difficult to design good levels. As a result, all maps were painstakingly constructed by hand. As the artists put the finishing touches on the landscapes, the programmers, who doubled as level designers, scripted the AI for the levels.

MYTH took approximately two years from start to finish. It began as a six-degree of freedom engine that allowed you to fly around a landscape. Soon, troops were added, heads started flying, blood was made to destructively alter the terrain's color map, and the network game was born. Most of the first year was spent developing the initial network/multiplayer game play. Almost the entire second year was spent developing the single-player game, refining the levels, and testing bungie.net, our free online service.

What Worked

1. BRINGING CARNAGE TO THE MASSES. It's a real trick to create a simultaneous, identical-look-and-feel, cross-platform release. It's even harder to do so within the expected time frame with only three programmers. Our experience porting MARATHON, our popular Macintosh-only action game, to Windows 95 was a valuable learning experience, and we vowed when starting MYTH that, "This time, we're going to do it right."

Doing it "right" meant designing MYTH from the ground up to be cross-platform compatible. Ninety percent of the code in the game is platform independent; the other ten percent is split evenly between routines that handle PC- and Macintosh-specific functionality. It was a programmer's dream come true… we spent almost all our time implement-

Jason Regier is currently a senior programmer at Bungie Software. He has four titles under his belt and was lead programmer for two of them. He started making games professionally in 1994 while attending Harvey Mudd College and has been doing so ever since. He can be reached at jregier@bungie.com and welcomes résumés from anyone interested in joining Bungie for future projects.



ing features and solving real problems, rather than wasting it fighting the OS.

All of the data for MYTH, from animated cut scenes to the percentage of warriors who are left-handed, is stored in platform-independent files called tags. Tags are automatically byte swapped when necessary and are accessed via a cross-platform file manager.

One of our programmers worked in conjunction with Apple Computer Inc. to develop a cross-platform networking library code-named Über. One of the greatest things about Über is that it supports plug-in modules for network protocols. Thus, although MYTH currently only allows games over TCP/IP, AppleTalk, and through TEN, it would be trivial to add support for new protocol modules. MYTH must provide a user interface to set up the connection, but once Über establishes that connection, game play over a LAN is the same as over the Internet.

To keep the game's appearance identical across platforms, we implemented our own dialog and font managers. This allowed us (actually, it required us) to use custom graphics for all interface items. We designed our font manager so that it supported antialiased, two-byte fonts, as well as a variety of text-parsing formats. Thus, our overseas publishers Eidos and Pacific Software Publishing were able to localize relatively painlessly. The German version of MYTH was finished only a couple of weeks after the English release, with Japanese and French versions close behind. The only game experience that is different for the two platforms is the installation, and two players on bungie.net have no idea whether their opponents are on Macintoshes or PCs.

2. BUNGIE.NET AND BETA TESTING. MYTH was also released with integrated support for our first online service, bungie.net. This service was designed specifically for MYTH and was developed simultaneously. Similar to online services for other games, it allows players to connect via the Internet to game rooms, where they can chat or play against one another. The Linux-based server that runs bungie.net keeps track of player statistics and gives everyone a score in our ranking system. The service's web site (www.bungie.net) has access to this database and sports a leader board that lists the top players.

Our networking layer is based on a client/server model. Once you advertise a game on the network, you become a server, and other players join your game. Network traffic during a game is limited to the commands issued by the players. All copies of MYTH in a network game run deterministically and merely interpret the commands that they receive. This makes cheating difficult; if you hack the game to perform something illegal, such as making all your units invincible, you'll go out of sync with other players. When portions of the game data are periodically checksummed and compared, a message will indicate that you're out of sync (and out of luck). So far, the only form of cheating we've encountered is users trying to exploit the bungie.net ranking system.

To rigorously test our server load capacity and the bungie.net code, we released a public beta of the network game. We were initially apprehensive because it was our first public beta test of a product, but it was an amazing success. When errors occur, MYTH alerts the player, logs the error messages, and usually allows the user to save a replay of the problem. Testers submitted these detailed bug reports via e-mail and chatted about features and improvements to levels on internal newsgroups.

Best of all, the testers used bungie.net to give instant feedback to the developers. This interaction allowed us to gather even more useful information about bugs, and it made the testers really feel involved in the final product. By the end of the beta-testing cycle, we not only had a clean product, but also had a loyal following of users who sang our praises when the NDAs were lifted.

3. 3D GRAPHICS ACCELERATION. When the project started, 3D acceleration hardware was only just starting to become popular. Nevertheless, we tried to keep hardware acceleration in mind when designing our rendering pipeline. When the opportunity arose to add hardware acceleration, the implementation worked beautifully. We worked closely with people from 3Dfx and Rendition and added support for their chipsets in about a week. It's amazing



how much these accelerators add to the smoothness of the terrain, the fluidity of camera movement, and the realism of the units and effects. These chips rock, and great on-site developer assistance made them easy to support.

4. Getting back to the people. Once we had released Myth, we definitely did the right thing by waiting for player feedback and then releasing a patch to address their issues. Since our public beta test caught most of the bugs in the shipping product, nearly all our post-shipping efforts were directed towards adding user-requested features. We scoured the newsgroups, read e-mail, and talked to customers about their complaints. From these disparate sources, we compiled a list of improvements for our 1.1 patch.

All major user complaints were addressed in the patch. We added support for Rendition and Voodoo Rush cards. We extended the camera's maximum zoom for a better view of the battlefield. We made our easy difficulty levels even easier. And we improved the unit AI. By the time the early reviews came out, we'd already released a beta patch that addressed almost everything on the reviewers' lists of Myth's failings.

5. Doing more with less. It doesn't take fifty people to create a major cross-platform software title. Period. Bungie Software has barely half that number of employees in the entire company, and we not only develop all our games, but publish and distribute them as well! Macintosh and PC versions of Myth, all our internal tools,

and our online service were essentially developed by only six people, and everything shipped on time with no major glitches. There's no big quality assurance department here at Bungie; the public did our testing for us, and we listened to them as seriously as if they were coworkers on the project.

We didn't hire any game designers, writers, or level designers to come up with our game concept and story line. Myth truly is the combined vision of our team, and each of us feels that it was our game. We came to work each day excited about the project, and we're damn proud of what we managed to create.

What Went Wrong

1. Staffing problems. On the flip side, it became clear very early in the project that we were understaffed for such an ambitious undertaking. Success or failure rested with a handful of people, and that was extremely stressful. Losing a programmer halfway through development added still more pressure during the final push to get the game out the door. Additional programming tasks had to be shouldered by the remaining developers, who were already also responsible for level design. To alleviate the problem somewhat, we even found it necessary to ask our busy network administrator to aid in AI scripting and level design.

We did hire a third artist near the end of the project, but it was almost too late. While his contributions to the final product were by no means insignificant, it took a long time to get him up to speed. Similarly, when we dropped the services of our original sound guy late in the development cycle, a new sound team had to rush to redo all the work.

If you're looking for good anecdotes about how we blew off steam with wild weekend trips to Cancún, you won't get any. We all worked incredibly hard, and did so willingly because Myth represented a two-year labor of love. All the great previews and supportive feedback from beta testers kept us excited and made us realize that we really did have something special on our hands. Nobody wanted to slack off and allow competing products to beat us to the shelves. The moral of the story: staff up as early as possible and plan to weather the unexpected.

2. SCRIPTING. The biggest announced feature that didn't make it into the final version of MYTH was a scripting language that would allow the player to modify elements of the game. We had hoped that user scripts could be written for extensible artificial intelligence, as well as custom formations, net game rules, and map behaviors.

We selected Java as a good basis for the MYTH scripting language because of its gaining popularity, good information-hiding capabilities, and relatively simple byte code interpretation. After several months of work, early versions of the game loaded, compiled, and ran code from tag files. A few simple scripts worked for presentation purposes, including one that instructed a unit to search the battlefield for the heads of the enemy and collect them in a pile.

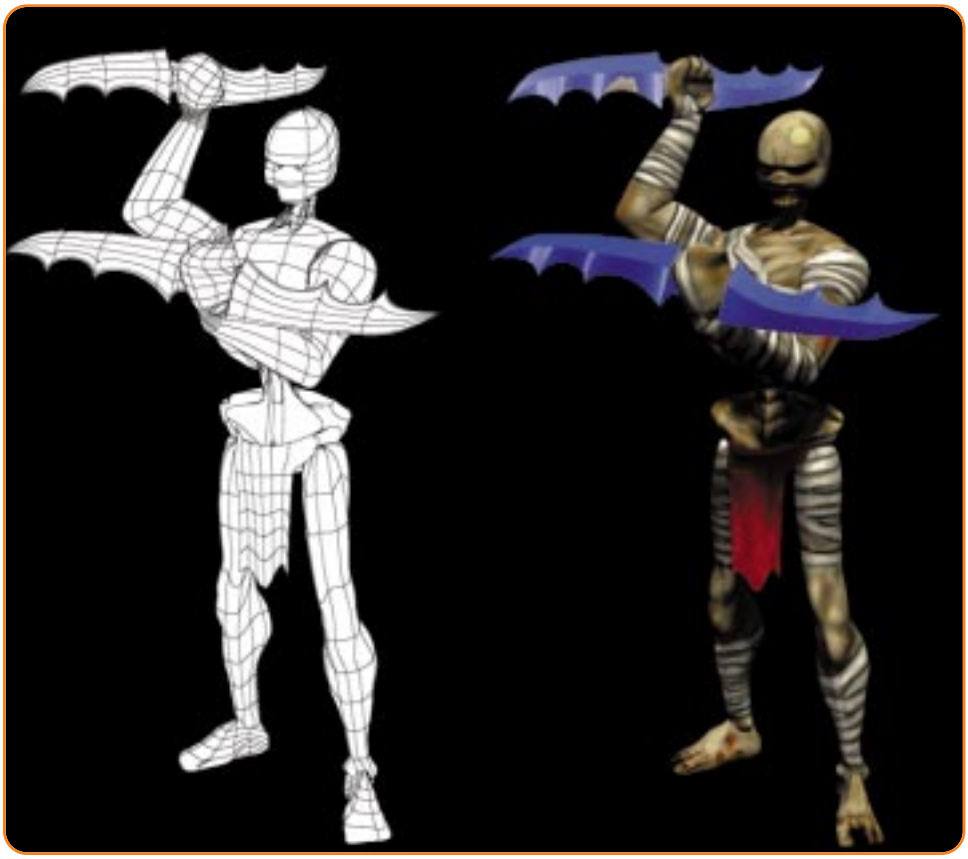
Unfortunately, when the programmer responsible for the scripting language parted ways with Bungie, we were left with a number of features to implement and no library of user-friendly interfaces with the game code. Given its incomplete state at such a late stage of development, there was little choice but to drop this functionality.

3. MORE FRAMES OF ANIMATION. One of the complaints most often voiced by players is that the sprite-based units' animations are not fluid enough. At the start of the project, when we planned for the number of frames of animation per unit, there was a good deal of uncertainty regarding how much RAM would be consumed by large texture maps, sounds, and other resources. As things were, it was not uncommon for our landscape textures to reach 5MB in size, and certain animations already consumed close to 1MB — our uncertainties were not unfounded. We erred on the conservative side. Though we implemented caching schemes that greatly reduced our memory requirements, there wasn't enough time to rerender the units.

4. PATHFINDING. Perfect pathfinding seems to have become the Holy Grail for games in the RTSG genre, and MYTH is no exception. The game's terrain is a 3D polygonal mesh constructed from square cells, each of which is tessellated into two triangles. Cells have an associated terrain type that indicates their impassability, and they may contain any number of solid objects, including trees, fence posts, and units.

Ah! Square cells, you say? Having read previous *Game Developer* articles (Bryan Stout, "Smart Moves: Intelligent Pathfinding," *Game Developer*, October/November 1996; Swen Vincke, "Real-Time Pathfinding for Multiple Objects," *Game Developer*, June 1997), your first thought may be that the A* pathfinding should do the trick. The first problem with a pure A* approach for MYTH is that impassable obstacles, such as troops and trees, may lie anywhere on the terrain. Penalizing the cells beneath impassable obstacles is a bad idea because the cells are fairly large and obstacles are not guaranteed to be aligned at the center of a cell. Furthermore, even if a tree did consume exactly one cell, the A* path to avoid it would make a unit walk up to the tree, turn, and continue around it. Units that bump into trees and walk between the centers of large cells appear extremely stupid; you really want your group of troops to avoid obstacles (including each other) ahead of time, and smoothly weave their way through a forest.

To produce this effect, we created our own pathfinding algorithm. First, we ignore all obstacles and calculate the A* path based solely on the terrain impassability. For all intents and purposes, the terrain in MYTH never changes, so this path can be calculated once and remembered. Now, we consider the arbitrarily placed obstacles and periodically refine



our path using a vector-based scheme. If the planned path would cause us to hit an obstacle, we need to deviate our path. We recursively consider both left and right deviations, and choose the direction that causes us to deviate least from our A* path. Thus, we've considered terrain impassability information and we can avoid arbitrarily placed (or even moving) obstacles well before we bump into them.

For every game, pathfinding is a pretty complex and sensitive beast. This method worked well for 90 percent of our cases, but rigorous testing revealed certain cases that were not adequately handled. As the ship date drew near, we were forced to say "good enough" rather than handle these problem cases and risk introducing new bugs. Our current algorithm works pretty well and provides the effect we sought, but there's definitely room for improvement.

5. FEATURES THAT MISSED THE CUT. With a few exceptions, everything from our list of "Stuff that Rocks" made it into the final product. Those features that didn't make it came so close and were so exciting that they definitely deserve mention.

Near the end of the project, we started adding support for 3D fire, which would be ignited by explosions and flaming arrows. Our flames were sprite-based 3D particle effects, complete with translucent smoke. Fire could spread across the landscape and move at different rates over the various types of terrain. To our dismay, when a spark in the woods spread into a raging forest fire (as it should), all the translucent smoke sprites slowed even fast, 3Dfx-accelerated machines to a crawl. With little time to rectify the problem, we had to put out the fire, so to speak.

We had also planned for wildlife to scamper across the terrain and for birds to fly through the air, breathing life into our empty landscapes. Our attempt at ambient life started with a giant squirrel created by one of our artists. Unfortunately, due to time constraints, we didn't have a chance to create very interesting behaviors for it. Just about the only AI that we had a chance to code simply made the squirrels gravitate towards the player's units. We thought it best to drop ambient life rather than subject players to hordes of nuzzling squirrels.

Post-Release Reactions

With all the prerelease hype MYTH had received, we were very anxious to see how the public would receive the final version. The reactions from beta testers were phenomenally positive, as were the comments from customers and reviewers. Our swiftness in correcting problems and adding several user-requested features with a 1.1 patch only earned us more kudos from the press and public.

But possibly the most satisfying result of the game is the degree to which it lessens the appeal of playing with a traditional isometric perspective. Working on MYTH so consumed our time that we didn't get a chance to play anything else; we looked forward to playing some old favorites and the latest demos of our high-profile competition after we shipped. It was a real surprise to discover that once we were accustomed to MYTH's 3D camera and its associated freedom, playing isometric games was frustrating — the action seemed distant and unrealistic, while the view of the world was annoyingly rigid. This sentiment was echoed in both player comments and reviews of the game. Since our MARATHON products were derided by some as DOOM rip-offs, it was especially satisfying to hear players say that MYTH pushes the genre in a new direction, from which there's no looking back.

As of late 1997, MYTH: THE FALLEN LORDS had shipped 350,000 copies worldwide in four languages on two platforms. bungie.net currently boasts tens of thousands of registered users and is being expanded to keep up with the constantly increasing demand. As I write, it has just been declared Game of the Year by *Computer Games Strategy Plus* and Strategy Game of the Year by *Computer Gaming World*. It remains to be seen whether MYTH will inspire other entries into the 3D real-time strategy game genre. But if nothing else, MYTH is proof that a very small team with a strong product vision can still make a very big game. ■

SOAPBOX

by Chris Hecker

Developer Power and the "U" Word

I'm going to talk about something I feel very strongly about. It's a subject that might get me laughed off the stage as some kind of wacko, or at best a hopeless idealist. We'll see.

Editor's Note: This was originally presented as a speech at the Seattle Computer Game Developer's Mini-Conference in November 1997. It has been edited for length.

This topic is much bigger than texture mapping, rigid body dynamics, or any of the technoid topics I've written about in the past. If you put my thoughts on this topic into product development terms, they would be in the pre-alpha stage, so bear with me, and don't expect a completely polished and airtight presentation.

I'm going to talk about power and control in the game industry.

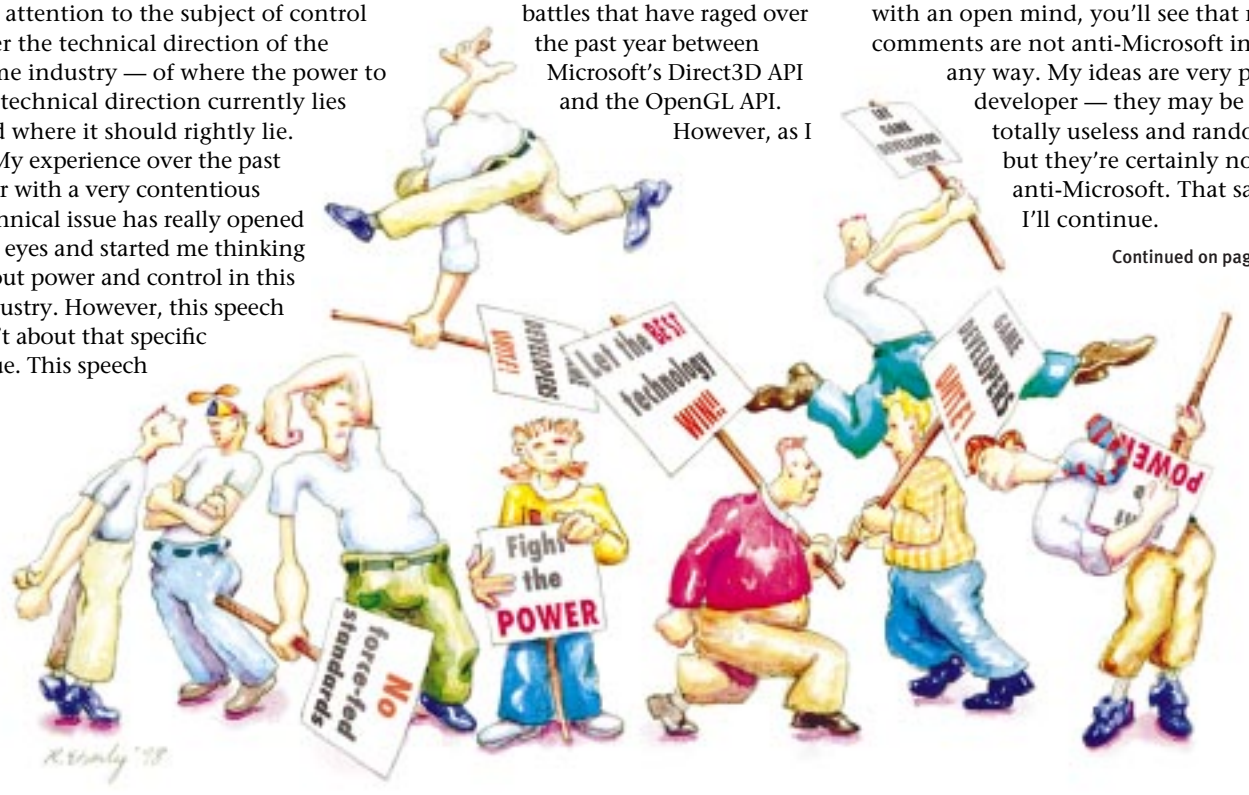
This is a huge subject that permeates every level of this industry, from the artistic to the financial. In the interest of staying focused, I'm going to restrict my attention to the subject of control over the technical direction of the game industry — of where the power to set technical direction currently lies and where it should rightly lie.

My experience over the past year with a very contentious technical issue has really opened my eyes and started me thinking about power and control in this industry. However, this speech isn't about that specific issue. This speech is about the meta-issue that was underlying it at all times, but was never brought to the fore and discussed in any meaningful way. Because I can't claim to have all the answers at this pre-alpha stage, I will simply bring the meta-issue into the spotlight where we can hopefully have an intelligent discussion about it.

The contentious issue that I'm referring to is, of course, the 3D API battles that have raged over the past year between Microsoft's Direct3D API and the OpenGL API. However, as I said, this speech is not about 3D APIs. Feel free to substitute in your own contentious technical issue (whether it's Sega's choice of 3D chip in their next-generation console, Nintendo's choice to use cartridges, Intel's MMX instructions, or something else), and I think the meta-issue will stay the same: who makes the controlling decisions and what are their motivations for making these decisions?

Before I answer this question, I feel compelled to give a sort of disclaimer. A couple of people have tried to paint me as being "anti-Microsoft," and this simply isn't true. It's a shame that I even have to mention this, but I think if you really listen to what I'm saying with an open mind, you'll see that my comments are not anti-Microsoft in any way. My ideas are very pro-developer — they may be totally useless and random, but they're certainly not anti-Microsoft. That said, I'll continue.

Continued on page 70.



Chris Hecker is the technical director of definition six inc. He can be reached at checker@d6.com.

Continued from page 72.

So, who makes the controlling decisions and what are their motivations for making these decisions?

With regards to my specific example of the 3D API battles, the answer to the "who" part of that question is Microsoft. Microsoft decided to make Direct3D the "official" 3D game API for Windows. Now, as for their motivations, we could formulate various theories, but as I said, this speech isn't about the 3D API issue. Beyond any specific motivations, what is the underlying motivation at the very base of all the decisions made? There only can be one answer for any well-run, publicly traded company. That motivation is raising the stock price. Of course, there are other secondary motivations (especially at the individual employee level), but at the base of it all, Microsoft, SGI, Intel, Nintendo, Sega, Sony, IBM, and all the rest must make decisions that will raise their stock price, or they'll be out of business.

Should we hate Microsoft for its decision regarding 3D APIs? Should we think that the company is inherently evil or engaged in some sort of Usenet conspiracy hijinks? No. I certainly don't hate Microsoft for making this decision. The company has a fiduciary responsibility to its shareholders, and business plans and technical directions undertaken by Microsoft must uphold this responsibility.

However — and this is the crux — just because something is a good direction for Microsoft or another large and influential company does not necessarily mean that it's a good direction for the game industry. Why should it be? They're completely separate entities. Sure, sometimes (and possibly, the majority of the time) directions taken by large corporations will be in line with what's best for the game industry, but sometimes they won't.

Which brings us to my thesis: it's our failure as game developers when a direction taken by an influential company is not in our industry's best interests.

How can it be our failure when we aren't the ones making the actual decision? Because we failed to stop that decision from being made. We failed to change the decision into something that is best for us as game developers, rather than best for that company's shareholders. Rather than the company's direction changing because we, its customers, demand it, we settle for a different direction because the company demands it. That is 100% backwards from the way it should be. Rather than the industry setting the direction, the company sets direction and the industry is only allowed to provide feedback on that direction — feedback that may or may not be heeded. The final decision lies with the company. Again, 100% backwards.

Remember the last time that you were sitting in your office trying to write to some awful API or instruction set, and you started ranting about how the person who designed it had clearly never written a game and how you could design a better API in your sleep? And remember how your coworker told you to shut the hell up and get back to work because that's the way things are in this industry? Well, guess what. You were probably right and your coworker was wrong.

The secret to turning this situation around is understanding where the real power in the game industry lies. The fact is that power lies in our hands as the creative force driving this industry. All of this money and politics and business exists because we make games that are fun to play and that people buy as a result. Without our games, the industry collapses. That's why we have the real power.

I've seen glimpses of this power in the past year. One time was when I organized the OpenGL letter from

Product Resources

Product	Company	URL	Phone	Page
3D Studio MAX	Kinetix	www.ktx.com	415-547-2000	55
3D Studio R4	Kinetix	www.ktx.com	415-547-2000	40
Animator Pro	Kinetix	www.ktx.com	415-547-2000	37
CodeWarrior	Metrowerks	www.metrowerks.com	512.873.4700	63
Dune 1.0	Nichimen Graphics	www.nichimen.com	310-577-0500	10
Game Control Interface (GCI)	Quantum3D Inc.	www.quantum3d.com	408-919-9999	10
Lucidity RT	Digital Media Interactive Inc.	www.dmix.com	650-655-4924	11
MeshPaint	Positron	www.3dgraphics.com	402-330-7011	38
Optimize	3D Connection	www.3d-connection.dk	+45 32 96 98 62	43
Photoshop	Adobe	www.adobe.com	408-536-6000	37, 63
PowerAnimator	Alias|Wavefront	www.aw.sgi.com	416-362-9181	40, 63
StudioPaint	Alias|Wavefront	www.aw.sgi.com	416-362-9181	63
TrueMotion 2.0	The Duck Corporation	www.duck.com	212-692-2000	11
Visual C++	Microsoft	www.microsoft.com	425-882-8080	59, 63
VTune	Intel	http://developer.intel.com	408-765-8080	57

game developers asking Microsoft to support OpenGL on Windows 95. Now, you might ask, what good did that letter do, since Microsoft didn't actually end up supporting OpenGL on Windows 95? [This was written before the joint Microsoft/SGI OpenGL announcement in December 1997, so maybe the letter did have some positive effect after all! - Ed.] But you would be overlooking the real effect the letter had. No, Microsoft didn't heed the letter's demands, but the letter did wonders for the morale of game developers interested in using OpenGL, and it let people in isolated pockets know they weren't so isolated in the industry after all. It showed IHVs that developers were serious about using OpenGL for games. And, of course, it brought the press spotlight onto the issue.

The reaction to the letter indicated to me that the power to change the direction of the industry for the better truly does lie in our hands. However, tapping into this power is extremely difficult for two reasons.

First, we only have true power when we act collectively, and getting people to think bigger than themselves or their current project or their company is very difficult. We need to start thinking about what's best for the industry globally and in the long run, in addition to how you can get these bugs fixed or make your milestone on Friday.

If we can't make that leap, then others who can think strategically (such as, say, a large, well-organized company) will control our destiny forever, and we should stop complaining about it. If we're not going to do something about it, then we forfeit the right to complain.

Second, even once people are thinking in this long-term and global mindset, it's incredibly hard to get them to spend any time doing anything about it. I often hear developers say, "Sure, I want to use OpenGL in the long run, but I've just got to ship this game before I can think about it." Well, there might not be an OpenGL in the long run if developers don't actively support it. This is true of any technical issue, not just 3D APIs. There will be a small window in time in which to act for the good of the industry. If we miss it, we'll live forever with whatever technology is being pushed, whether it's the best technology for our industry or not.

Let's make a flying leap and say that I've convinced you that we really do have this power, and you're now thinking globally about various issues that affect your daily job and this industry as a whole. Where do we go from here? I think the raw energy to harness the power is already here and being spent on a daily basis. Every time you rant about some brain-dead technical direction in which we're being led, that's energy that we could theoretically apply to fixing the problem. Most developers rant like that every day, and that's a lot of energy that's just dissipating. We need to focus it.

One way to focus that energy is to form ad hoc interest groups, as I did for the OpenGL issue. I got on the phone and e-mail and organized a ton of game developers, and one of the results of that organization was the letter. This effort worked and it's still working, but it's incredibly tiring. There would be days when I was supposed to be working on our game, but instead I'd spend all day on the phone or answering e-mail. I care deeply about this issue, but I can't let it ruin my company, so I'm forced to budget the time that I spend on it. I like to think that if I could have worked full time at organizing people, there wouldn't even be an issue anymore — we'd have won already. However, there's no way I could spend all of my time on it, so I'm sort of forced to do a half-assed job if I want our company to survive.

I'm not the only one who's tried to organize ad hoc technical committees. Zack Simpson of Titanic Entertainment, who used to be at Origin, tried twice to do this — once with sound standards a few years back, and again with joystick APIs. Like me, Zack found that it takes an incredible amount time to organize people and get results.

One could argue that the reason that it takes so much effort to organize people on these issues is because people really don't care, but I don't think that's true. People do care, and if someone's doing the dirty work of organizing (such as me, Zack, or someone else), then developers are willing to spend a bit of their own energy. It's just that there's such a huge busywork barrier to entry for setting up this leverage that it rarely happens.

So, I think the ad hoc organization isn't a viable model in the long term. I think that we need a more persistent body, one where the initial costs of setting things up can be amortized across a number of issues: a body that has respect so that these companies pay attention to its demands, and a body that has teeth for when the companies don't pay attention.

Now, this is the part where you want to get the rotten tomatoes ready. I'm going to use the "U" word. Union.

Now, of course I don't mean a union in the full Jimmy Hoffa-sense of the term, since this isn't a labor-management situation. Mostly, I use the term union because it seems to have some mystical power these days: people either hate them or love them. Hopefully, I got your attention.

However different our situation may be, if you look at some of the words and concepts that I've been using, such as "a group of people who are powerful together, and powerless when separated," "collective bargaining," and "putting teeth into our demands," you start to see some similarities. If you don't like the term union, you could call what I'm talking about a standards body with the means to enforce its standards.

This would be a body made up of people who have a vested interest in seeing the absolute best thing done for the industry, and that means us developers. If we were well-organized, we could easily force companies to do the right thing, rather than hoping and praying that they do the right thing, as it is now. The teeth we would wield would be everything from simple endorsements, logo programs, and press releases (you'd be surprised by how much these large companies fear bad press!) all the way to boycotts or working with a competing vendor to shut out a disagreeable company.

So now what? How do we move forward? This is where it becomes evident that my thoughts are still in the pre-alpha stage. I can think of a number of organizational models, such as companies pitching in money to fund the organization and developers volunteering their own time. I will think about this some more, and I encourage you to think about it as well. I think the time has come for this idea, but it's not going to happen unless we make it happen.

And now for the requisite quotations:

Voltaire said, "The best is the enemy of the good."

This quote is apt for our situation, but John Miles rephrased it recently for the Direct3D/OpenGL battles, and I like his version even better.

Miles said, "The good-enough is the enemy of the excellent."

I don't know about you, but I'm sick of settling for good-enough when it's well within our power to have excellence. ■