gd

GAME DEVELOPER MAGAZINE

MARCH 2000

UBM Tech

Roll Your Own ILM

Last October in my column ("Graphics Fly...Will Developers Fry?") I said that unless game developers are willing to live with significantly longer game development life cycles in the future, there may soon be shortfall between what games *could* look like and what they *will* look like. Tomorrow's consumer hardware — both PCs and consoles — will be so powerful and capable of supporting such realistic visuals, that (barring some unforeseen technology that solves this problem) it may take scores of artists to satisfy the scene complexity requirements of the typical game. Moore's Law is writing checks that artists may soon be unable to cash.

One hope that I put forward as a possible remedy was for better development tools to help create and manipulate art assets. Those tools may or may not appear, but even if they do, I'm beginning to doubt whether they alone will solve all of our problems.

If better tools alone don't help, it's likely that companies will throw more artists, modelers, and animators at the problem. That solution might generate some of its own problems, though. The cyclical, project-focused nature of game development implies crunch periods and lulls over time. Some small and medium-sized companies won't find it cost-efficient to keep a large number of artists and animators on the payroll year-round. (Not to mention the fact that some small game development studios *want* to stay small.) So increasing staffs significantly might not be the answer, either.

If sufficient tools don't materialize, and hiring an army of artists isn't an option, it seems that outsourced modeling and animation is going to become a much bigger part of our lives.

The film industry faced a similar problem in the late 1970s after *Star Wars* came out and ushered in the era of big-budget, special-effects-laden movies. Other movie studios wanted to create the same high-quality effects that Lucas had, but some came to the conclusion that their core competency wasn't in the field of generating special effects, and turned to visual effects houses like Industrial Light & Magic. The visual effects industry was born.

The game industry seems to be on a similar trajectory. To adapt to the larger content demands of tomorrow's games, it seems likely that many of tomorrow's projects will have to rely on contracted work on a large scale. You can already see the beginnings of this shift. Motion capture studios such as House of Moves and Locomotion have done well by the game industry and continue to thrive. Stock and custom 3D models by Viewpoint Digital are used by scads of companies. And of course you can't overlook the hundreds of individuals and small firms like 3D Pipeline and Etribe Studio that provide 3D animation services.

I think we're just around the corner from a consolidation in this portion of our industry. Digital Domain and ILM have already dabbled in some game projects, and they're in a good position to capitalize on the growth of the game industry. Unquestionably, some of today's smaller consulting companies will grow and expand their offerings. And although it's more of a long shot, imagine a large game company like Electronic Arts spinning off segments of its business into a full service visual effects company, just as Lucas did with Lucas Digital.

Managing the massive flow of assets arriving from an outsourced army of highly trained illustrators, modelers, animators, video editors, and other artists and creative technicians will mean new challenges to the in-house producers and art directors that opt to send their art needs outside. It will probably shake up the way companies structure their art paths. The makeup of some game studios might change, weighting staffs more heavily with programming and design talent. If such events unfold, many of us will feel the impact. ■

ON THE FRONT LINE OF GAME INNOVATION
Game DEVELOPER
600 Harrison Street, San Francisco, CA 94107
t: 415.905.2200 f: 415.905.2228 w: www.gdmag.com

Publisher
Jennifer Pahlka *jen@mfgame.com*

EDITORIAL

Editorial Director
Alex Dunne *adunne@sirius.com*
Managing Editor
Kimberley Van Hooser *kvanhoos@sirius.com*
Departments Editor
Jennifer Olsen *jolsen@sirius.com*
News & Products Editor
Daniel Huebner *dan@mfgame.com*
Art Director
Laura Pool *lpool@mfi.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Jeff Lander *jeffl@darwin3d.com*
Mel Guymon *mel@infinexus.com*
Omid Rahmat *omid@compuserve.com*
Advisory Board
Hal Barwood LucasArts
Noah Falstein The Inspiracy
Brian Hook Verant Interactive
Susan Lee-Merrow Lucas Learning
Mark Miller Group Process Consulting
Paul Steed id Software
Dan Teven Teven Consulting
Rob Wyatt Microsoft

ADVERTISING SALES

National Sales Manager
Jennifer Orvik e: *jorvik@mfi.com* t: 415.905.2156
Account Representative, Silicon Valley
Mike Colligan e: *mike@mfgame.com* t: 415.356.3486
Account Executive, Northern California
Dan Nopar e: *nopar@mfgame.com* t: 415.356.3406
Account Executive, Western Region
Darrielle Sadle e: *dsadle@mfi.com* t: 415.905.2182
Account Executive, Eastern Region
Afton Thatcher e: *athatcher@mfi.com* t: 415.905.2323

ADVERTISING PRODUCTION

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

MILLER FREEMAN GAME GROUP MARKETING

MarCom Manager Susan McDonald

CIRCULATION

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

INTERNATIONAL LICENSING INFORMATION

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

Miller Freeman
A United News & Media publication

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

BPA INTERNATIONAL

BIT Blasts

News from the World of Game Development

New Products

by Daniel Huebner

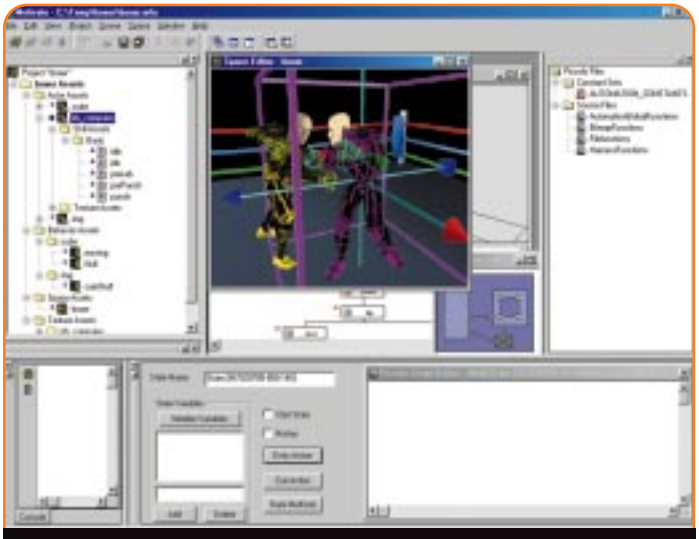
A Character-Building Experience

THE MOTION FACTORY is readying version 2 of its Motivate Developers Toolkit for shipment early this year. The Motivate Toolkit is a real-time animation and behavior programming toolkit designed to enhance character realism while cutting production time.

Motivate's animation engines and scripting language allow for easy creation of believable, interactive 3D characters and environments. The system includes development tools, an SDK, server technology, and multiple run-time engines. The redesigned core architecture for this version is compact and modular, enabling developers to select, replace, or omit parts of the system to fit specific needs. Motivate 2 adds new real-time articulated body dynamic simulation and expanded multi-platform and Internet support. Developers can also use the Motivate SDK to extend and customize the system.

Optimized run-time engines are available for Windows, Playstation 2, Dreamcast, and Macintosh. The Motivate 2 Developers Toolkit will be available sometime in the first quarter. A complete package including all tools and engines will be priced a $7,500 per seat, with a commercial run-time distribution license fee of $50,000 to $100,000 per title. Pricing for individual engine modules, including DLLs and the SDK, run from $17,500 to $50,000.

■ The Motion Factory
Fremont, Calif.
(510) 505-5151
http://www.motion-factory.com



Motivate 2 has redesigned its core architecture allowing developers to tailor the system to their individual needs.

More Power to the Modelers

TGS has released version 5 of Amapi 3D, its NURBS and polygonal modeler. Improvements over previous versions include dynamic geometry, new tools for smoothing and deformation, new display modes, and polygon reduction technologies.

Amapi's new dynamic geometry can remember the construction history of complex surfaces so that the user can dynamically edit an object by changing its outline, profile, or structure. Smoothing tools in version 5 offer four unique modes while the new deformation tools allow instantaneous object bending, twisting, and tapering. Amapi 3D also offers a 3D silhouette display mode to simplify complex scene management and speed the modeling process. Amapi exports into 25 file formats, including new support for NeMo, Cinema 4D, and Zap.

Amapi 3D 5 is available for Windows 95/98 and for Mac OS 8 and higher at a list price of $399. Existing users can upgrade for $199.

■ TGS Inc.
San Diego, Calif.
(858) 457-5359
http://www.tgs.com/amapi

Resoundingly Easy Facial Animation

LIPSINC has developed Echo, an automated character lip-synching system for a wide range of animation packages that reduces the entire process of lip-synching to a single step in the overall production schedule.

The basis of Echo is Lipsinc's VoiceDSP voice analysis system. Echo uses this digital signal processing technology to analyze speech and automatically output the proper mouth, jaw, and lip position data in three dimensions. Echo outputs flipbook, dope sheet, and function curve animation data for various animation platforms, 3D game engines, and multimedia applications. Output data is generated as a stream of precise morph targets that accurately re-create lip-synched animation.

Full and evaluation versions of Echo for Windows 98/NT are available though the Lipsinc web site. A technical support package is included with each seat of Echo, and the company also offers consulting to customize Echo's output. Licenses start at $10,000.

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

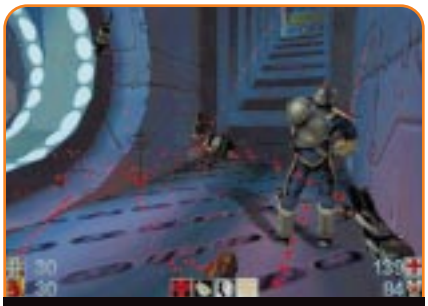
Industry Watch

by Daniel Huebner

MATTEL HIRES STOLAR. Former Sega of America head Bernie Stolar will again attempt to change the fortunes of a struggling videogame power. Mattel has announced that Stolar will be the new president of Mattel Interactive and will be charged with solving the problems currently plaguing Mattel's interactive toys and software division, including stemming the losses of Mattel's troubled 1999 acquisition of The Learning Company. Stolar, who left Sega of America after disputes with its Japanese parent company over strategy for the American Dreamcast launch, will be a key part of Mattel CEO Jill Barad's overall plan to reshape the company. Mattel also announced the retirement of CFO Harry Pearce, who spent 24 years with Tyco before joining Mattel in a 1997 merger. Pearce's departure is likely also related to financial problems resulting from The Learning Company acquisition.

FULL-TIME JOBS. In his keynote address to the Macworld masses assembled in San Francisco in January, Steve Jobs reported that he is dropping the word "interim" from his title and will become Apple's official CEO. In addition to unveiling Apple's new operating system, OS X, Jobs announced that Apple is investing $200 million in a partnership with Internet service provider Mindspring to create a Macintosh-branded ISP. The new ISP will be the default Internet service on all Apple computers, leading Jobs to speculate that Apple could become one the world's top Internet companies. Unfortunately for Macintosh gaming evangelists, a demonstration of QUAKE 3: ARENA running on Apple's new OS X promptly locked up the system.

BRAZIL BANS SIX GAMES. In response to a recent shooting in Brazil linked by the media to videogames, Brazil's Justice Ministry issued an order banning several game titles. The ministry labeled DOOM, MORTAL KOMBAT, REQUIEM, BLOOD, POSTAL, and DUKE NUKEM as being too violent for sale in Brazil. Police were ordered to remove all copies of the offending titles from store shelves, and vendors failing to comply



REQUIEM: under fire in Brazil.

have been threatened with a fine of nearly $11,000 a day. "The games are considered violent [and are] affecting people who play them, particularly children," said a government spokesperson. "[DUKE NUKEM] may have motivated [alleged gunman] Mateus da Costa Meira to stage the cinema shooting on November 3 in São Paulo." Brazilian police believe the shooter was reenacting a scene from the game. The ministry plans to issue a ruling in the near future on other games considered excessively violent.

GAMES.COM PREPARES FOR LIFTOFF. Hasbro is gearing up to launch its online games portal Games.com and has chosen Go2Net as its technology partner. Hasbro is looking to launch the game service sometime in the middle of this year, and plans to offer no fewer than 50 titles encompassing Hasbro properties such as Atari, Wizards of the Coast, and Microprose. The majority of the planned games will be multiplayer Java versions of classic games like Monopoly, Clue, and Battleship. Go2Net will, under the terms of its three-year agreement with Hasbro, build the site and provide technology for chats, messages, and community management. Hasbro will gain access to games on or in development for Go2Net's Playsite gaming service, while Go2Net acquires the rights to Hasbro's most popular games for use on a co-branded site, as well as $7.5 million fee from Hasbro. Hasbro plans to spend $60 million in the next year to launch and develop Games.com, and hopes to expand the site eventually to six separate game channels focusing on family, kids, arcade, game shows, sports, and hard-core players.

BLEEM! BATTLE RAGES ON. The court battle between Sony Computer Entertainment and Playstation emulator developer Bleem! took a turn when a U.S District Court judge granted Bleem!'s motion for leave to amend and assert counterclaims against Sony. Bleem! claims that Sony unlawfully acquired, maintained, and extended its monopoly in the videogame market through a combination of anti-competitive practices, including misuse of copyright, patents, and other intellectual property. The judge also blocked Sony's request to modify a protective order covering Bleem!'s confidential business data. Sony had sought access to source code as well as reseller and customer information. "Sony simply has no business demanding this kind of proprietary data from innocent third parties, particularly when it has no bearing on the case whatsoever," said Bleem! lead attorney Jon Hangartner. ■

UPCOMING EVENTS CALENDAR

GAMEXecutive Conference 2000

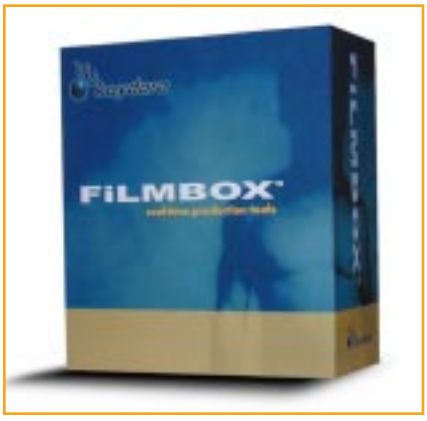
MONTEREY CONVENTION CENTER
Monterey, Calif.
March 7–8, 2000
Cost: $1,300–$1,900
http://www.gamexecutive.com

Game Developers Conference 2000

SAN JOSE CONVENTION CENTER
San Jose, Calif.
March 8–12, 2000
Cost: $200 and up
http://www.gdconf.com

NAB 2000

LAS VEGAS CONVENTION CENTER
Las Vegas, Nev.
April 8–13, 2000
Cost: $150 and up
http://www.nab.org/conventions/nab2000



Kaydara's Filmbox 2

by Jeff Lander

Creating animation for real-time 3D characters has become a major ordeal. In the days when character animation was based on sprites, your characters would have animation loops of typically six to ten frames in five to eight directions. You needed a few people working in DPaint or Photoshop, but the cleanup work wasn't too bad. Tools like DeBabelizer made the job downright streamlined.

These days, however, smooth real-time animation is the rule of the day. A game such as Sega Sports' NFL 2K contains more than 1,500 animations, each of which comprises anywhere from a few frames to hundreds. This amount of animation data is becoming increasingly common and if you're in charge of making all those animations look right, you're going to be one busy little artist. Whether you decide to have animators create the motion for your characters or you use some form of performance capture to generate the initial animation, there is still a lot of work to be done.

Many producers are under the impression that performance capture provides game-ready animation in a nice, clean package. Actual users of this technology, however, know this is not the case. Motion data needs to be cleaned up, trimmed, and massaged to fit your actual characters. Some of this can be done by a performance capture service provider, but there is always some work that must be done in-house.

Likewise, producers may wish to save money by using the same animation for multiple characters in the game. They may also want to blend multiple animations together to create transitions. Both of these things are possible, but only by carefully manipulating the animation data. Most professional animation packages allow you to work with the animation data to some extent, but as these packages also strive to provide many other functionalities, a modeling and animation program may not be the best tool for the job.

ENTER THE FILMBOX. Kaydara designed Filmbox specifically to fill this role. Animation manipulation is its specialty. Filmbox provides a complete suite of tools for capturing, creating, editing, and blending animation data. It also provides support for a variety of devices for inputting motion data as well as real-time playback and rendering options. However, at its core, Filmbox is about the motion.

When you first start up the program, you will immediately notice it has a distinctly different feel, as you can see in Figure 1. The interface clearly shows the program's SGI-based roots through its nonstandard interfaces. Early on, I spent quite a bit of time trying to find buttons described in the manual. It turned out that the buttons were at the bottom of a scrollable window. It was not at all evident to me that the window could scroll and it took me a while longer to figure out how to scroll the window, as there were no scroll bars. It turned out that clicking the left mouse button plus shift and dragging on the window was the secret. Having used several SGI programs, learning these methods was not a big problem for me, but for those of us more accustomed to Windows applications, it can be a bit frustrating. Fortunately, there is a Quick Reference Guide that explains the various key-mouse combinations and their functions. However, unlike the previous version of the program, there are no longer quick-start tutorials. This plus the lack of an index in the reference books can make it intimidating to the beginning user. The program also clearly prefers running in 1280×1024 mode as things seem to fit better. However, that was not clear to me from the manual.

HOW DO I USE IT? Filmbox allows users to import a great variety of animation data. It supports the most thorough list of animation formats that I have ever seen. This includes the main motion file formats such as .BVH, Acclaim,

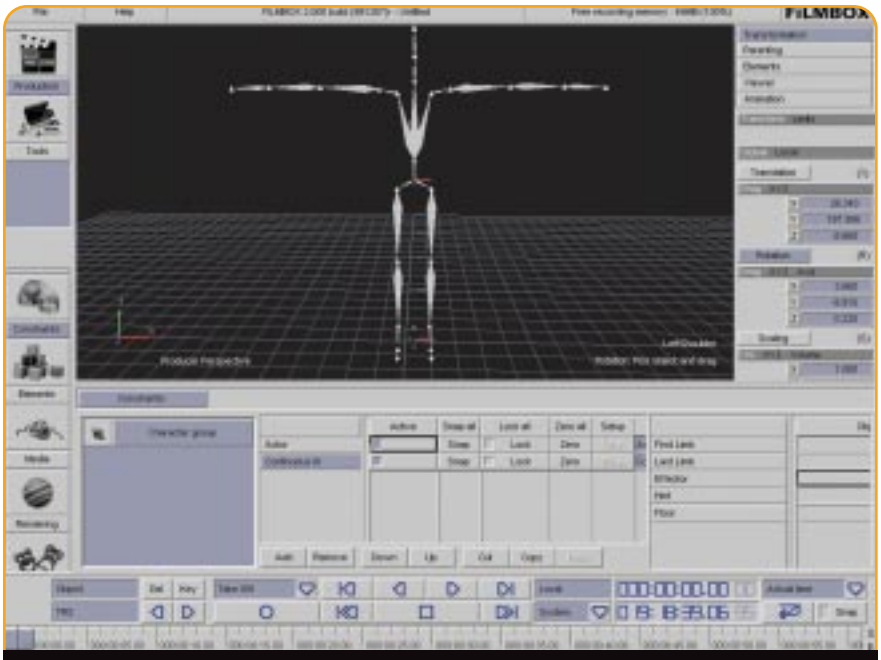


FIGURE 1. *Filmbox's IRIX-inspired interface may be intimidating to Windows users.*

Jeff is trying to figure out how to attach ping-pong balls to his cats to capture some nice quadruped motion. So far, he is just getting scratched up. If you've got some better moves, e-mail them to jeffl@darwin3d.com.

Character Studio, and Motion Analysis's .TRC, among others. You can also bring in animation and models from 3D Studio Max, Maya, Softimage, and Lightwave. Once the animation is imported, you can manipulate it in many ways.

One of the main problems with the use of a performance capture device is that it can create too many keyframes in the animation stream. This is due to the fact that a capture device simply generates a keyframe for every point of capture every frame. Much of the cleanup work that a capture house does is removing keyframes that are not needed and eliminating spikes in the data where the hardware had a glitch.

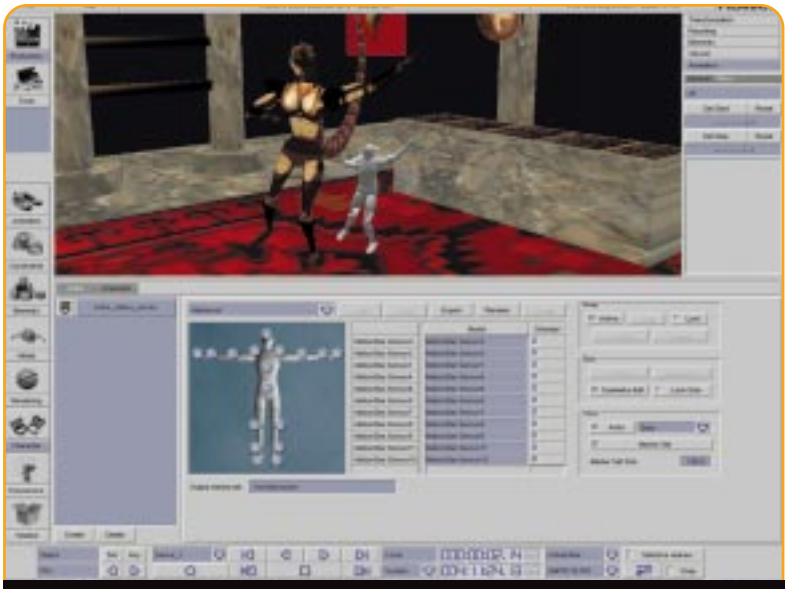


FIGURE 2. *The Character tool with motion applied.*

A typical motion file from a capture session can generate so many keyframes that it may be too much data storage for your game. Using Filmbox, you could resample the data from 30 frames per second to 10FPS. But a more powerful option is to fit a Bézier curve to the data and apply control points as needed. This can reduce the keyframes significantly while retaining the original curve of the animation channel. There are also tools for spike removal, as well as for time shifting and curve scaling. These tools go well beyond any provided by standard animation and modeling packages.

Another key step for working with animation is applying the motion to your characters. Filmbox has one of the most intuitive methods for making this connection. Using the Character tool (Figure 2), you create a link between a logical "actor" created in Filmbox and your own character model. You can bring in your character from any modeling package and relate the joints in the character with the joints in the Filmbox actor. This process is

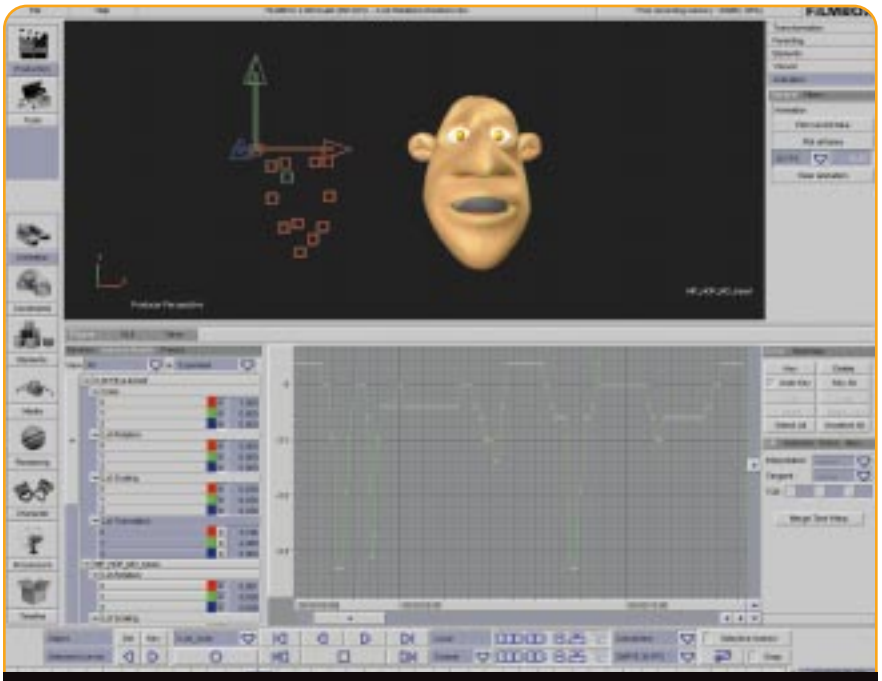


FIGURE 3. *Shape animation aids deformation effects such as facial animation.*

streamlined for biped characters but is quite easy to set up for any character hierarchy once you get the hang of the process. Once this association is set up, you can apply any motion to your character and start manipulating it.

BEYOND THE BASICS. Once you start digging deeper into the program, you will begin accessing the more advanced features. For one thing, you can set up a great variety of constraints to the character. This includes the typical stuff such as look-at, positional, degree-of-freedom, and IK constraints. However, there are also more sophisticated, world-aware features such as a "floor contact" constraint, which attempts to keep the feet in contact with the floor, and an "enforce gravity" constraint, which keeps the hips between the feet. There are also interesting options called "reach hands" and "reach feet" constraints. These attempt to get your character to match the position of the hands and feet of the actor, regardless of the motion, which is useful when trying to apply motion to characters of different sizes. When the motion calls for a character to reach out and grab something such as a door handle, you want the character to hit that mark no matter how large you make him. With these constraints you can scale the motion up and the actor will still hit the mark.

The package also includes a sophisticated expression language which allows you to create a complex mathematical expression to control the animation, which appears to be more robust than many similar expression systems I have used.

Beyond working with motion, there are tools to control the real-time playback of your performance such as real-time camera animation and switching, a robust set of lighting and shadow controls, and texturing tools. Filmbox also supports shape animation (Figure 3), which I find very useful for facial animation and other deformation effects. Like the skeletal animation system, these effects can be completely controlled through external input devices such as MIDI controllers or even audio files. You can also purchase the optional Voice Reality module which will convert an audio input into visemes automatically, though it currently supports only five visemes. There is also an optional rendering module to display your creation with a Cartoon Reality Shader for real-time performances.

For anything you find lacking in Filmbox there is the SDK extension system which allows you to build your own features into the package.

WHAT'S THE BOTTOM LINE? It would take a user a vast amount of time to go beyond scratching the surface of all of Filmbox's features. It's a very robust system for working with and manipulating motion data. It is this robustness, however, that tends to narrow its usefulness. Filmbox must work with an external package for creation of the characters. Users will need to have a fairly powerful animation package to make Filmbox really useful in production. For many users, at the cost of many high-end modeling packages an additional art tool to manipulate the animation data will be unnecessary no matter how powerful it is.

That said, if your project requires you to manipulate and adapt a great deal of animation data and your regular art production tool is not up to the task, Filmbox will definitely provide you with all the power you need. I am certain that with the great number of animation-intensive titles coming along, many will find this an indispensable tool in their developer's arsenal. ■

Filmbox 2: ★★★★

Kaydara Inc.
Montreal, Quebec
(514) 842-8446
http://www.kaydara.com

Price: Filmbox Animation: $4,995; Filmbox Matchmove: $9,995; Filmbox Motion Capture: $11,995; Filmbox Online: $29,995

System Requirements: Windows NT/2000 or IRIX, 128MB RAM, fast OpenGL acceleration (suggested), 3D modeling/animation package.

Pros:

1. Sophisticated manipulation tools for motion capture and animation data.
2. Ability to blend animations together to create transitional moves.
3. Import/export plug-ins for most 3D animation packages.

Cons:

1. Nonstandard user interface will take time to learn.
2. Product is highly targeted to fill a very specific role in production. May not be useful to you unless your project requires a lot of animation processing.
3. Lack of tutorials and a manual index can slow learning curve.

Shades of Disney: Opaquing a 3D World

Like many people who work with computer images, I am a huge fan of classic animation. The amount of labor that goes into creating an animated feature film has always amazed me. Consider for example Disney's *The Jungle Book*. The film is 78 minutes long. At 24 frames per second, that's 112,320 frames of animation. Each of those frames needed to be drawn, cleaned up, inked, painted, aligned with the background, and finally shot to film. Each of these steps required a great deal of skill and patience on the part of the artists involved.

Look at the job of the opaquer. This person is responsible for receiving the final cels from the ink department and coloring them using opaque paint. This job is essentially coloring in between the lines using a paint-by-numbers key known as a color model. While it seems like a fairly straightforward — though repetitive — job, opaquing was very time consuming during the early years of animation. Shamus Culhane, who was deeply involved in the process at Disney for many years, estimated that his opaquing department could average about 25 cels per day. At that rate, it would have taken his team 12 years to opaque the cels for *The Jungle Book*. Clearly, the staff for this Disney classic worked their little animated tails off.

Fortunately for the animation industry, computers have come along. Through the use of a digital ink and paint system, a single artist can opaque several hundred cels in a single day. As an extra benefit, the computer eliminates many of the problems artists had matching colors painted on various layers of acetate. Painting an animated feature is still a major issue in animation, but the job has gotten much easier.

Enter the Next Dimension

In the digital world of 3D real-time animation, I have some opaquing problems of my own. Last month I looked at methods for creating cartoon-style rendering on 3D objects. I was able to deal with creating the silhouette and material lines, however I had yet to get the cartoon look for the surface of the object. I suggested that I would need to look to texturing techniques to get that part of the job done.

You can see the situation I would like to end up with in Figure 1. Given one light shining on the model, I want there to be a clear separation of the light and dark halves of the model. A classic model for illumination gets me most of the way. I want the shade to be a function of the surface normal and the light position. In the Lambertian reflection model, the brightness of a surface position depends only on the angle between the direction to the light source, *L*, and the surface normal, *N*. Mathematically, that would be

$$I = k_d(N \bullet L)$$

The dot product is taken between the surface normal and the light source direction and is multiplied by a diffuse lighting constant. Since the dot product will vary from 0 to 1, this would just give me the basic Gouraud-shaded-

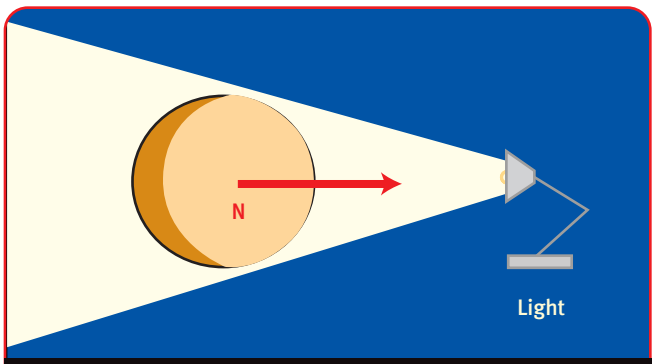


FIGURE 1. *A nice cartoon shading showing a clear delineation between light and dark portions of the model.*

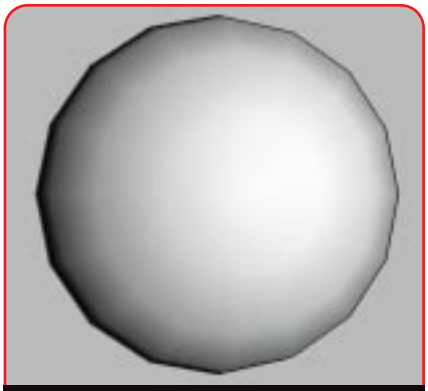


FIGURE 2. *A plain Lambertian reflection gives a Gouraud-shaded look.*

When not glued to his TV watching the Cartoon Network, Jeff can sometimes be found at Darwin 3D. Send him a message at jeffl@darwin3d.com and we will slip it to him during a commercial break.

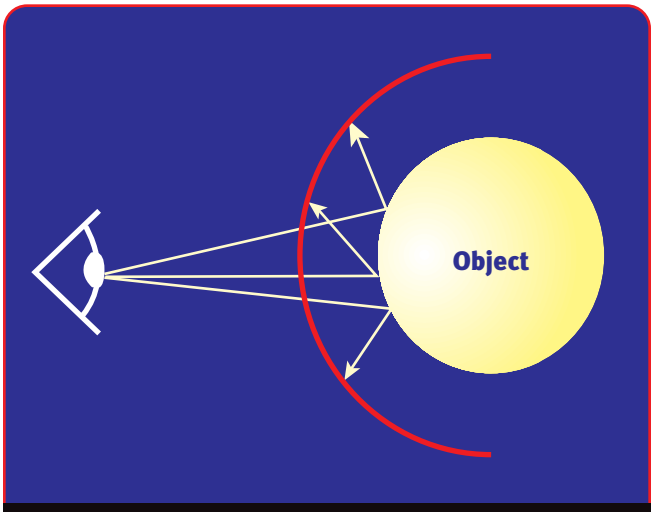


FIGURE 3. *The spherical environment-mapping technique can be used to make an object look as if it is reflecting light.*

ball look where the illumination value goes smoothly from white to black as you can see in Figure 2.

What I need to create is a cutoff where the illumination is "light" or "dark." The ideal formula would be

$$I = \left(k_d \left(N \bullet L\right) < \varepsilon\right)$$

where ε is the shading threshold. As I discussed last month, I could compute the vertex colors at every vertex using this formula. However, this wouldn't get the desired results. Graphics hardware interpolates the vertex color across each triangle. Since the cutoff point could potentially occur in the middle of a triangle, a simple interpolation would not look correct.

It's tempting to consider using environment-mapping techniques to create the effect. Spherical environment-mapping calculates the ray from your eye that reflects off the surface to the point that it strikes on a hemisphere around the object. You can see this illustrated in Figure 3.

This technique is used to make things look reflective, like shiny metal. It's also a method for creating a specular highlight on an object. I could create an environment map that transitions from light to dark, as seen in Figure 4, then apply that to the object. This gives me exactly the result I was looking for but has a few problems. For one, the coordinates are calculated from the eye point. In order to get the look I want, I will need to calculate the environment map from the light. This is possible, but kind of a pain.

The second problem is that if I wish to change the shading threshold or the number of in-between values, I would need to recalculate the environment map completely. That would be a bit of a burden on the CPU.

Using a Texture as a Lookup Table

Another thing you may have noticed is that the map in Figure 4 is a bit wasteful. The same color gradient is repeated around the circle radiating from the center. Let's look again at the formula I am trying to reproduce. I know that the dot product term will vary from 0 to 1. I can calculate the value for *I* for each dot product from 0 to 1 and store it in a table.

$$I = \left(k_d \left(N \bullet L\right) < \varepsilon\right)$$

$$\sum_{u=0}^{1} ShadeTable[u] = \left(k_d \left(u\right) < \varepsilon\right)$$

For example, suppose $\varepsilon = 0.375$. The table would look like Figure 5.

Now I can take this table and convert it into a one-dimensional texture (I know you've probably always wondered how those could be used). I set up the 1D texture in OpenGL with a couple of easy function calls that are almost identical to their 2D equivalents.

```
glGenTextures(1,&m_ShadeTexture);
glBindTexture(GL_TEXTURE_1D, m_ShadeTexture);
// Do not allow bilinear filtering
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, m_ShadeWidth, 0,
GL_RGB , GL_FLOAT, (float *)m_ShadeSrc);
```

You will notice that I turned off filtering. This is because I actually want to get a banded, shaded look. If filtering were on, the colors would be blended in a way that would not look at all right for my purposes. Since filtering can slow things down on some cards, this ends up being beneficial for performance as well.

In order to use this new 1D texture, I simply need to calculate the dot product and index that result into the table as a texture-map coordinate. For an object that can rotate, the

FIGURE 4. *A cartoon environment map, transitioning from light to dark.*

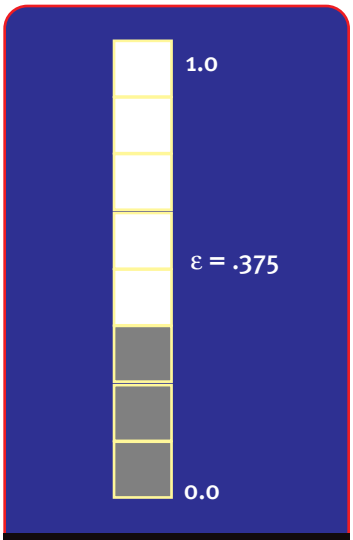


FIGURE 5. *A lookup table for the given value.*

vertex normal will need to be rotated by the object matrix before the dot product is calculated. The code for calculating the table index is given in Listing 1.

This new lookup map is applied just like a normal texture map except for the fact that it is 1D and requires only one texture coordinate. The results of this process can be modulated with the object's surface color to get the final image. This gives me a great deal of flexibility in how the shadow is applied across the image. I can control the cutoff level and the number of levels of shading across the surface, and I can even add a highlight by brightening the top of the table. To recalculate the values, I only need to update the table — quite a bit easier than the entire 2D texture map. The table can be of any resolution your graphics card can handle. If you use too few shades, the resulting surface will appear very banded and blocky. I found that for most objects, a 32-pixel table looks pretty good. You can see a variety of shade tables and their respective results in Figure 6.

LISTING 1. *Code for calculating the texture coordinates.*

```
////////////////////////////////////////////////////////////////////////////
// Function:  CalculateShadow
// Purpose:   Calculate the shadow coordinate value for a normal
// Arguments: The vertex normal, Light vector, and Object rotation matrix
// Returns:   An index coordinate into the shade table
////////////////////////////////////////////////////////////////////////////
float COGLView::CalculateShadow(tVector *normal,tVector *light, tMatrix *mat)
{
//// Local Variables ///////////////////////////////////////////////////////
   tVector post;
   float dot;
////////////////////////////////////////////////////////////////////////////
   // Rotate the normal by the current object matrix
   MultVectorByRotMatrix(mat, normal, &post);
   dot = DotProduct(&post,light);  // Calculate the Dot Product

   if (dot < 0) dot = 0;           // Make sure the Back half dark
   return fabs(dot);               // Return the shadow value
}
```

I now have a fast and flexible real-time cartoon renderer. The whole concept of using the texture-mapping capabilities of 3D graphics hardware to apply arbitrary functions across a surface is very powerful. You can create a very complex and completely nonlinear shade table and then apply it to the surface and let the hardware interpolate it.

Other Methods

Obviously I'm not the only person exploring the use of non-photorealistic techniques for real-time rendering in games. Sim Dietrich of Nvidia has been exploring the use of hardware-accelerated transformation and lighting for non-photorealistic rendering. Methods such as my use of the normal and dot product require the CPU to perform calculations on each vertex. Sim's goal is to minimize the use of the CPU by using features found on Nvidia's GeForce 256 GPU.

The GeForce 256 supports cubic environment mapping and texture-coordinate generation. By using a cubic environment map and the D3DTOP_DOTPRODUCT3 operation to generate texture coordinates for the environment map, Dietrich can create a cartoon rendering with very limited CPU impact. In addition, by applying more rendering passes, he can add some texture to the shaded part of the image. You can see some examples of Sim's work in Figure 7. On hardware that supports texture-coordinate generation and features such as cubic environment mapping, these methods are definitely worth exploring.

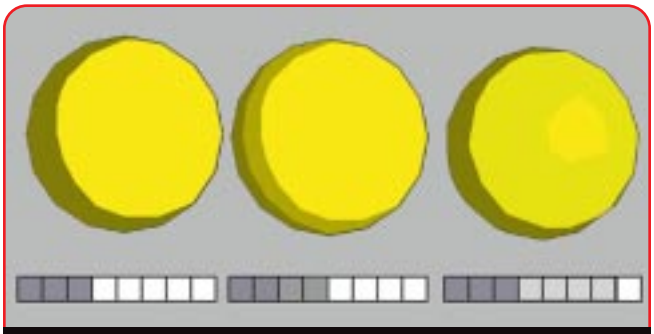


FIGURE 6. *Some shade tables and results.*

Intel Goes to Toontown

Intel has been creating a variety of impressive technologies available to the game development community. They have been working on a licensable real-time non-photorealistic rendering algorithm as part of the Intel 3D Software Toolkit that they are announcing at this year's Game Developers Conference. The software allows you to specify line settings such as thickness, color, and type. For the shading, you can set the shadow cutoff level and brightness as well as a highlight level and value. Intel has also been working on

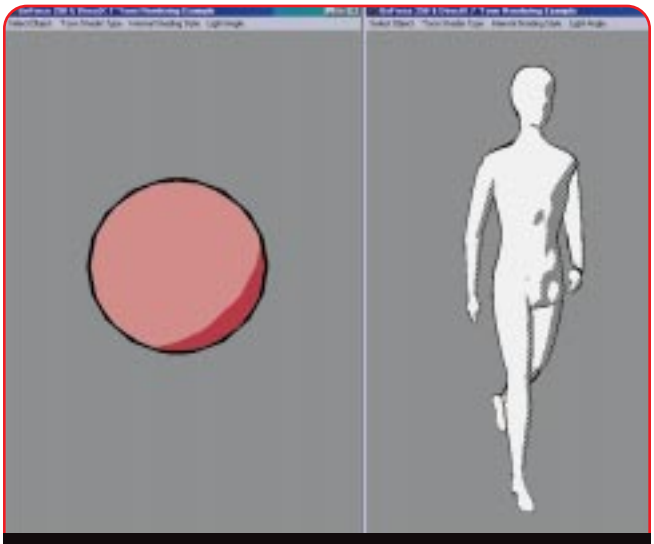


FIGURE 7. *This Nvidia cartoon rendering optimizes the GeForce 256 architecture, minimizing CPU impact.*

creating a variety of rendering styles such as sketch and pen-and-ink to go along with the cartoon rendering.

The 3D Toolkit will integrate this renderer with other 3D technology such as the multi-resolution mesh, subdivision surfaces, and a skeletal deformation system. The package will be available for use in a variety of real-time projects.

The Intel 3D Software Toolkit integrates a number of 3D technologies.

A Word About Digital Cinematography

Of course, even after all this work, creating the rendered look for your characters is only part of the battle. You also need to know how to display them effectively. In many real-time 3D games, the camera is tied directly to the character. It bobs along behind the character bumping off walls (or going through them in some cases) and wedges itself in places guaranteed to block the thing you are trying hardest to see. Camera control has become a major part of the 3D game creation process. Poor camera operation and control is sometimes enough to cause a game to receive a poor rating in game magazines. Clearly it's time to start thinking about the camera as an integral actor in the scene. Perhaps games have matured to the point that it's time to look to film production techniques and assign cinematographers to camera control in 3D interactive games.

Not long ago, most cinematics in games were created using traditional filmmaking and animation methods. These movies could break the immersive experience by pulling the player out of the action. These days, however, it seems like more developers are creating their cinematic sequences using the game's real-time engine. This trend has brought a variety of new problems with it. Many of these games use pre-scripted sequences for camera control to display the action in a pleasing way. This is fine for noninteractive sequences or dialogue trees. But if we want to have truly interactive sequences that the players can enjoy the way the project director intended, we need to take a serious look at the art of real-time camera control.

Consider the example of the action/adventure game. Many of these games use a tethered camera under complete user control. Game designers must be content letting the player manipulate the camera in order to show the action. Anyone who has played a game like this knows it can be very difficult to manipulate the character and the camera at the same time. Many times the player will get the camera into a "good enough" position and continue with the action, but this position will probably not be the best one for displaying the action.

One alternative approach I have seen is never to give players control of the camera in the first place. This can be frustrating to players as they may have a different idea of what is important in an interactive situation. Then there are the hybrid methods which yank control away from players to show them something "dramatic." This can be jarring and totally pulls players out of the interactive experience, leaving them no longer in control. It is clear to me that the interactive medium requires some fresh thinking about storytelling.

Filmmakers have been telling stories with the visual medium for a long time now. They have certainly learned a few things along the way. Out of those experiences a certain visual style has formed that guides basic filmmaking. I am not saying that these rules are not or should not be broken. However, when they are broken it is to achieve a desired effect, not simply out of ignorance of their very existence. This cinematic style, sometimes called continuity style, describes shot framing and staging methods that enhance storytelling without confusing audiences.

Next month, I'll be looking at methods for shooting an interactive story. Till then, think about the best and worst camera control you have seen in a game and let me know about it. ■

FOR FURTHER INFO

Traditional Animation

Culhane, Shamus. *Animation from Script to Screen*. New York: St. Martin's Press, 1988.

This book is an excellent source for all aspects of traditional animation — a must-have for animators and pretty useful for technical types. Covers everything from the walk cycle to facial expressions to starting your own studio. The only book I know that describes how a dodo bird walks.

Nvidia

http://www.nvidia.com

Sim Dietrich should have posted his document and application for cartoon rendering by now. If not, drop Nvidia's developer support an e-mail.

Intel 3D Software Toolkit

http://www.intel.com

Watch for a major announcement at the Game Developers Conference, March 8–12, 2000.

Skin Deep: Surfacing Strategies for Real-Time 3D Characters

For the past half-decade, the world of real-time 3D has been dominated by the humble yet ubiquitous polygon. Animators and artists in the gaming community have stood jealously by while their counterparts in the film and entertainment industries have honed and perfected their modeling techniques with NURBS and other exotic surfaces. Now, with Sega's Dreamcast flying off the shelves and the Japanese launch of Sony's Playstation 2 imminent, the face of RT3D gaming is in the process of being redefined. The hard-edged polygonal appearance is becoming a thing of the past, replaced instead by the smooth-surfaced and highly complex models that are, or will become, the standard fare for the next generation of RT3D platforms. Everything comes with a price, though, and to pay for this quantum leap in performance, we as developers — and especially as artists — must reinvestigate and possibly redefine our production techniques. And nowhere is this more apparent than in the construction and surfacing of the creatures and characters that populate our virtual worlds. This month, I'll discuss some ways artists can apply the latest technology to create more complex and more scalable character geometry.

Polygonal character modeling has long been the mainstay of RT3D gaming. The techniques for creating a model from the ground up, working only with its vertices and faces, represents the simplest, most straightforward method of creating geometry. Since the inception of RT3D development, it has been this inherent simplicity that makes the method so powerful. When the project has a limited polygon budget, heavy emphasis must be placed on polygonal efficiency, and it is mandatory that the artist get up close and personal with each and every vertex in a model.

With the latest trends in hardware, the scene complexity of RT3D games is skyrocketing. Scenes in excess of 50,000 polygons are soon to become the norm, and the pressure is on to create increasingly realistic content. This precipitates the need for greater complexity in character models, which will be sporting smooth organic surfaces. As a result, the same simplistic polygonal modeling techniques that were once so effective now become handicapped by their simplicity.

To understand the problem better, let's consider a character model that has 5,000 polygons, with level-of-detail (LOD) models comprising 3,000, 1,000, 500, and 300 polygons respectively. That's a total of almost 10,000 polygons' worth of modeling, just for a single character. Imagine the work involved in texturing and animating this character. Now multiply that by

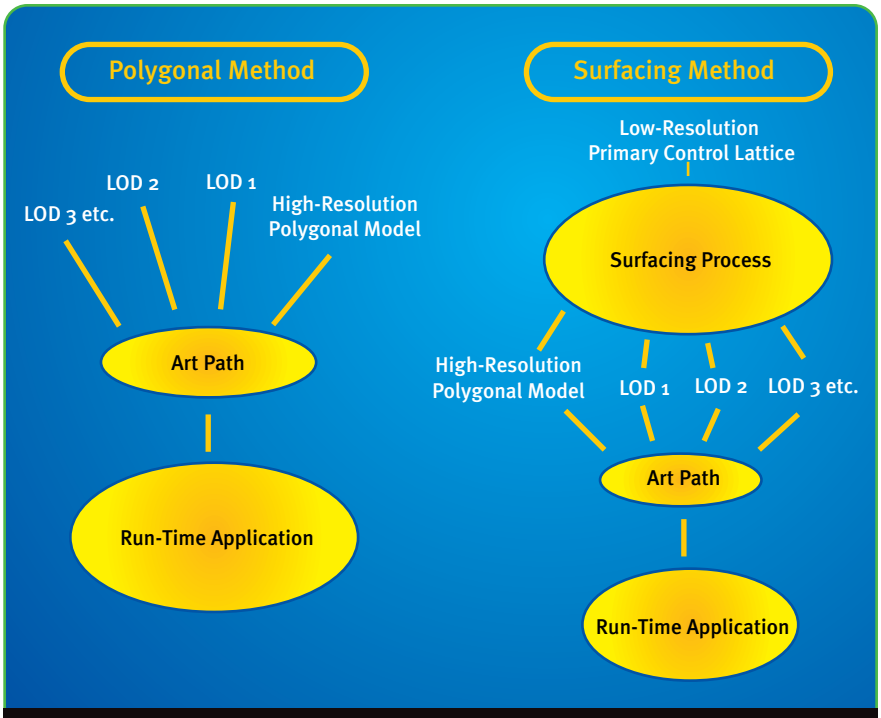


FIGURE 1. *With the surfacing method (right) artists need only deal with a primary control lattice, rather than multiple iterations of traditional polygonal models (left).*

Mel Guymon has been animating in the gaming industry for several years. When he's not at his desk pushing polygons, he can usually be found at the local Barnes and Noble, slumming for reference materials. Mel can be reached at mel@infinexus.com.

the dozens of characters which could potentially populate a RT3D environment and it soon becomes apparent that the standard polygonal methods for manipulating geometry rapidly break down. To a large extent, this is due to the fact that the modeling, texturing, and weighting techniques (applying vertex weights to a skeletal system) come with an interface that allows the artist to work directly with each vertex. Clearly, to avoid becoming bogged down by the sheer amount of data involved, we must augment or replace the standard modeling techniques with a higher-level editing method, one that gives artists the same degree of precise control without mandating that they work directly on the polygons.

The solution to this challenge is to come up with a method that allows artists to work with a low-detail control object which overlays the polygonal model. By working with this "primary control lattice" to the exclusion of the polygonal surface underneath, artists can keep their work scalable and resolution-independent. As a result, each aspect of character generation (modeling, texturing, and animating) becomes more efficient, and regardless of the model's final complexity and/or the number of LODs generated, the development time can be kept to a minimum (see Figure 1). So, if you think you might be ready to step beyond the bounds of standard polygonal character-generation methods, read on for an examination of the pros and cons of several alternate techniques.

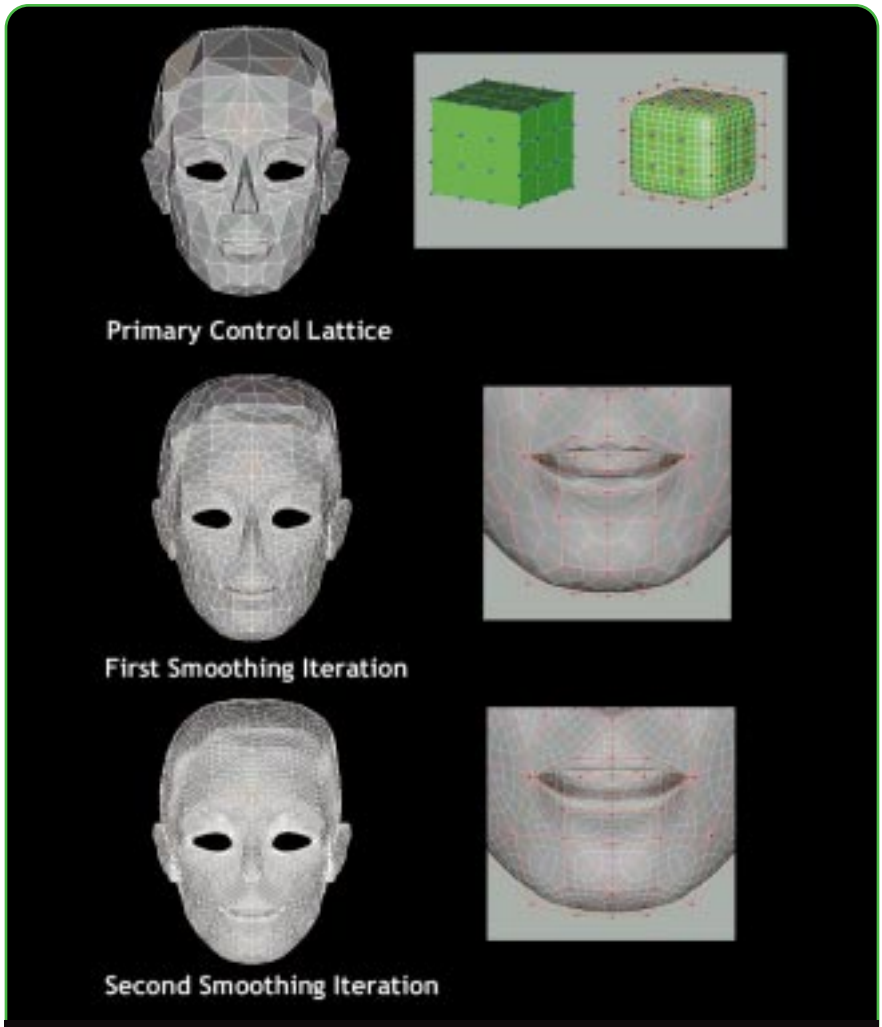


FIGURE 2. *The surface subdivision method requires artists to maintain only a small amount of data to generate the relative complexity of the final result.*

Surface Subdivision

The subdivision method augments the standard polygonal modeling techniques by simply subdividing an existing mesh to add complexity and smooth out hard edges. This method, shown in Figure 2, applies a modifier to the object that alters the surface of the model while leaving the original primary vertices intact. Information such as UV coordinates, vertex colors, and bone weights remain baked into the model as the mesh tessellates, allowing the modeler to work with a relatively small amount of data compared with the final result.

WORKFLOW. This method deviates the least from the standard polygonal method. First, the modeler creates a relatively low-resolution polygonal model (top left). This model becomes the primary control lattice, and all of the subsequent editing will deal with the vertices that make up this surface. In this case, the primary control lattice is polygonal, and as such, it will serve directly as the lowest level of detail for the model. Once the control mesh is created, a smoothing modifier is applied, and the number of iterations used determines the final in-game surface complexity. A first-order and second-order iteration smoothed model is shown on the left, and in the close-ups on the right, the vertices of the primary control lattice can be seen.

ADVANTAGES. This method is extremely straightforward, and since normal polygonal techniques are used to begin with, there is almost no learning curve. It is fast and easy to preview the result within the editing software, making it easy to iterate on the final result. Additionally, the primary control lattice for the mesh need not be overly complex to get a good result; this means a constant, relatively low overhead for the number of data points the artist has to work with. Finally, since the subdivisions are totally procedural, the smoothing algorithm can be applied in-game rather than in the editing software, with similar, predictable results. (See "Subdivision Surface Theory," January 2000, and "Implementing Subdivision Surface Theory," February 2000, for detailed information on this technique.)

DISADVANTAGES. First, because the primary control lattice is polygonal, the amount of information that can be stored in it is limited. In order to define

a curved surface, there needs to be at least three vertices. This means that for high-detail regions (such as a character's face), the control mesh still needs to be fairly dense. Second, the variation between levels of detail is severe. With each new iteration of the smoothing algorithm, the model's complexity can increase by a factor of four or more. Finally, the in-game surface may be constructed of uniformly irregular polygons, such that optimal rendering techniques (such as tri-strip rendering) are not possible, forcing the model to render more slowly than necessary.

AVAILABILITY. The technique as shown in Figure 2 was generated with the MeshSmooth modifier in 3D Studio Max 3, however similar functionality exists in Maya Unlimited's Surface Subdivision tool and Softimage's MetaMesh Extreme. Regardless, as I indicated above, the technique may be done procedurally within the engine, either at run time or as a post-process on the data.

NURBS

Working with NURBS (Non-Uniform Rational B-Spline) surfaces is a complete departure from the polygonal method. Long the mainstay of Alias|Wavefront's modeling suites, this surfacing technique has been gradually making its way into the standard toolsets of many of today's editing tools. A NURBS surface is a polygonal approximation of an object's volume as defined by a set of analytically generated curves. These curves, or splines, are further defined by a set of control vertices that can lie either on or around the curve. As you can see in Figure 3, the number of control points used is analogous to the number of exponents in an algebraic equation. The more convoluted the curve, the higher order the function is needed to describe it, and consequently, the more control points are required.

WORKFLOW. Modeling in NURBS surfaces requires a mindset different from that which one uses working with polygons. Although the final result is polygonal, at no time does the artist work directly with the polygons involved. At the most basic level, a NURBS surface is assembled by constructing contour curves of the object to be created (see inset, top left). These contour curves and the control vertices that define them will serve as the primary control lattice for the object (bottom left). Once the contours are complete, a surfacing tool is used to stitch the curves together. As in the previous example, UV coordinates, lighting information, and bone weights are accessed by modifying the control vertices in the primary control lattice. The resulting resolution-independent NURBS surface is shown, followed by two resultant polygonal equivalents.

ADVANTAGES. By far the biggest advantage of working with NURBS is that the technique has been in existence for a relatively long time. The tools have had time to mature and there exists a large user base of artists and animators who are familiar with this editing technique. Furthermore, though not as simple to execute as subdivision surfaces, the method is fairly straightforward, with the better editing tools requiring only a moderate learning curve. And although there are currently engines in development that will support NURBS shapes running in real time, if your particular engine requires polygonal models to be used, applying a NURBS-to-polygons modifier is a fairly uncomplicated procedure. Finally, if your engine does support NURBS, the on-the-fly nature of the polygonal subdivision routines makes it almost totally unnecessary to generate LOD objects.

DISADVANTAGES. Since the curves that make up a NURBS surface are analytical in nature, the resulting shape can be computationally expensive to work with. As a result, you may need a fairly robust workstation to work with a NURBS object of the complexity you require. Additionally, because the artist is not in direct contact with the resulting polygonal surface, the level of control over that surface is more limited than in the previous case. The surface regularity of the resulting mesh can also be unpredictable, depending on the application. As with the previous technique, this can make for less-than-optimal rendering performance.

AVAILABILITY. Again, this example was generated from a 3D Studio Max 3 tutorial, but similar functionality also exists within most of the major editing packages.

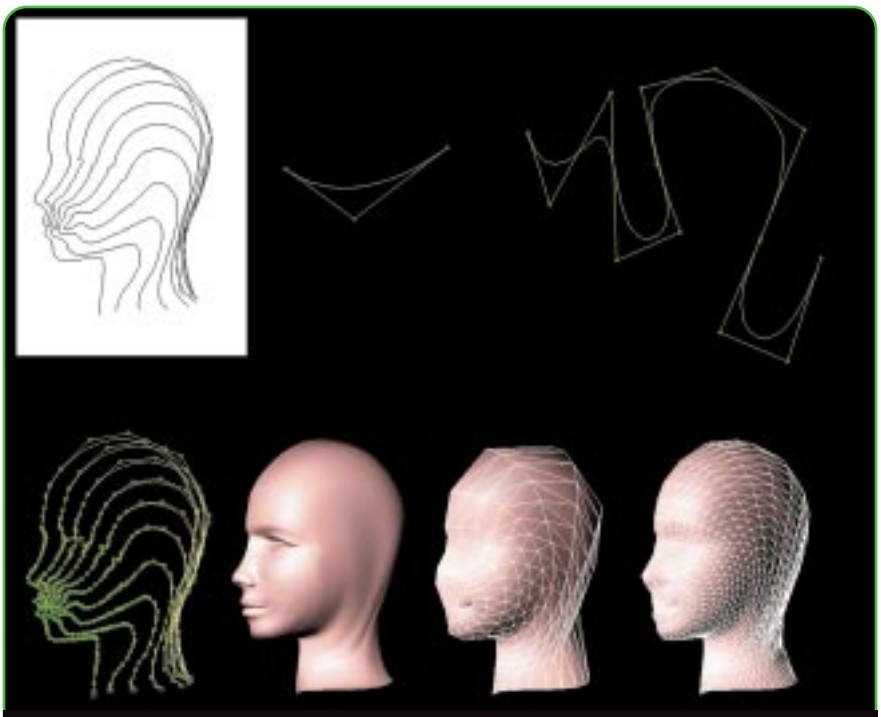


FIGURE 3. *NURBS can eliminate the need to generate LOD objects, but can result in unexpected surface irregularities and inferior rendering performance.*

Bézier Splines and Patches

A Bézier spline is a curve defined by two or more control points, each of which has two points that control

the tangent of the curve into and out of each control point. A Bézier patch is an editable surface defined by four control points and the edges or curves in between them. As you can see in Figure 4, the Bézier spline can store a lot of information in only two control points (top left). This is because the tangent controls for each point allow the artist to refine the curve further without adding additional points. The same holds true for a Bézier patch. Working with patches, artists can exert a large degree of control over the surface without worrying about having to add additional control points to the mesh. (For more information about Bézier curves and patches, see "Implementing Curved Surface Geometry," June 1999, and "Optimizing Curved Surface Geometry," July 1999.)

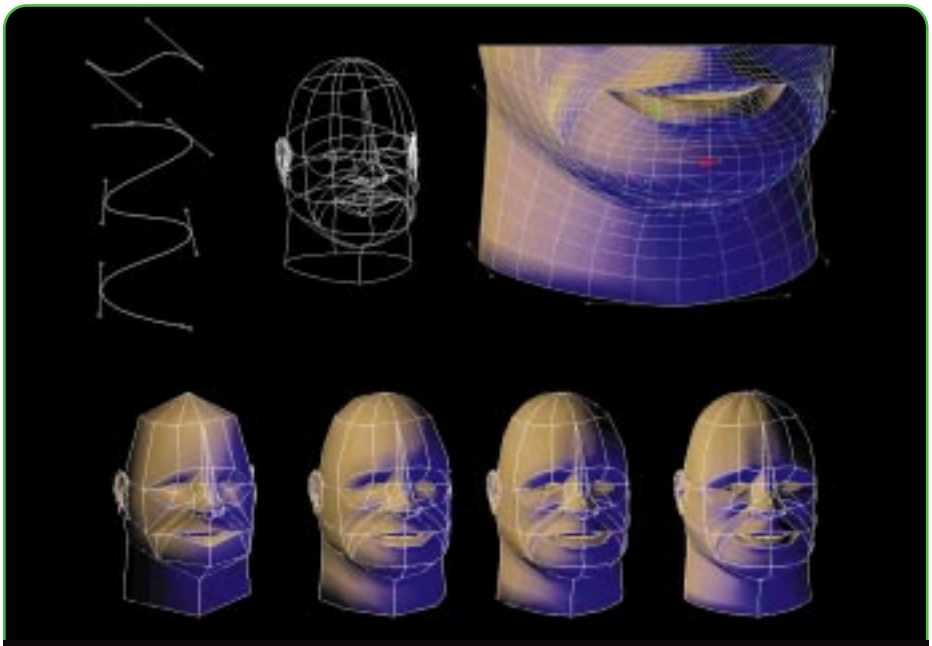


FIGURE 4. *A Bézier patch can be created using only two control points (upper left), with iterations on the surface increasing (at bottom) from left to right.*

WORKFLOW. The resulting method is somewhat of a mix between the two previous methods. Just as in the NURBS example, the object is first created by a series of contour curves (top center). However, these contour curves must run in both the U- and V-axes of the object. Where three or four curves intersect to form a three- or four-point shape, a patch can be defined. Once the primary control lattice is defined, a patch-surfacing modifier is applied to create the polygonal equivalent of the object. As with the first example of subdivision surfaces, the number of iterations on the surfacing modifier controls the complexity of each patch, and ultimately the complexity of the model. Unlike the NURBS example, however, if the number of iterations is set to zero (no curvature, shown on the model at lower left), each set of three or four control points defines a polygon that lies on the surface of the model. As with the previous two methods, editing the vertices in the primary control lattice enables the user to control UV coordinates, vertex coloring, and skeletal weighting. Along the bottom, the number of iterations on the surface increases from left to right. In the detail inset at the top right, the vertices and tangent handles of the primary control lattice can be seen.

ADVANTAGES. The patch approach offers a fast and easy solution with the best aspects of the previous two methods. Having the tangent handles on each control point is an extremely powerful tool, allowing the artist to add definition and detail without compromising the simplicity of the primary control lattice. As a result, the data storage in this primary control lattice is the most efficient of any of the methods I have discussed. Additionally, the polygons constructed using this method are extremely regular in appearance, resulting in a mesh that is highly optimized for rendering calculations. By increasing the number of iterations on the surfacing modifier, the complexity of the resulting mesh rose smoothly, yielding more control over the levels of complexity in the model. Finally, rendering routines for patch surfaces are becoming more widespread in use, and because of the regular polygonal construction in the surface, they tend to render faster than their polygonal counterparts.

DISADVANTAGES. The resulting primary control lattice requires a little more forethought to construct, since it's possible to create situations where more than four curves intersect, preventing the shape from becoming a valid patch. This can be extremely frustrating when first learning the technique. Also, the patch surfacing method is relatively new, resulting in a lower level of knowledge and experience in the development community as a whole.

AVAILABILITY. The patch method in this incarnation was demonstrated using 3D Studio Max 3. Somewhat similar functionality can be found in Softimage 3.8 SP2.

Find the Method That's Right for You

In the end, once I had experimented with all three techniques, I decided in favor of the patch-based method. It has the flexibility and power required for most character-based applications, and the resulting surface model is optimized for fast and efficient rendering. Regardless of the method you choose, however, the important thing to remember is to try to minimize the amount of data that artists have to work with. This way, the time saved in data manipulation can be spent on artistic finesse and game play, which is really what this whole show is all about. ■

Acknowledgements

Special thanks to Kris Renkewitz, Wyeth Ridgway, Dave Coathupe, and the folks who put together the 3D Studio Max 3 tutorials.

Infogrames: Today the Smurfs, Tomorrow the World

While the game industry likes to pat itself on the back in light of the stellar growth it has experienced in the last five years, I think it's worth singling out Infogrames as an exceptional example of growth in its own right. Like its French counterparts Ubi Soft and Titus, Infogrames has benefited from the interest of its home capital markets in interactive entertainment, using its financial muscle to draw Gremlin Interactive in the U.K., along with Accolade and GT Interactive in the U.S., into its corporate web. Infogrames is a prime example of the type of consolidation that is taking place in the industry, and an interesting study in how truly global the industry has become.

History

The outspoken Bruno Bonnell and Christophe Sapet founded Infogrames in France in 1983. Both were computer engineers who had worked in the electronics industry prior to getting into the game business, and like most engineers who follow the game route, they were PC developers first. Probably the turning point for the company came in 1989 when it became the first French company to obtain a license to develop games for Nintendo's SNES console. It was a piece of fortuitous timing, coming hot on the heels of the explosive growth of the SNES. However, the biggest prize for any real game company remained the lucrative North American market.

In 1992, Infogrames established I-Motion to build a presence for itself in the U.S. Despite its successes, Infogrames moved cautiously in its first forays into the U.S. Initially, Infogrames relied on Broderbund and THG for distribution, and up until the mid-1990s, more than 90 percent of the company's revenues were coming from Europe. Europe was a safe haven for Infogrames, and the U.S. market was very difficult in the early 1990s. For example, for many years Infogrames was a notable developer of Game Boy titles, and the company had carved a nice niche for itself in this market in Europe. In the early part of 1998, Infogrames sold more than 130,000 copies of its Game Boy title THE SMURFS, even though it was at least four years old at the time.

The North American Invasion

Co-founder Bonnell has gone on record to say, "We have targeted the U.S. market, and intend to replicate in the North American continent the success that has made us a leader in Europe."

In real terms, the company's initial goal was for the U.S. market to contribute $150 million to Infogrames' revenues by 2001. Recently, the company stated that it wanted its U.S. sales goal for fiscal 2004 to be $2 billion. As a result, a key event for Infogrames last year was raising $222 million through a five-year convertible bond offering in order to provide the company with a war chest to fund strategic acquisitions, primarily in the U.S.

Infogrames acquired Accolade in April 1999. This came in the wake of the acquisition of U.K.-based Gremlin Interactive a month earlier. In one fell swoop, Infogrames snatched up strong development teams in the U.S. and U.K., added 18,000 U.S. sales outlets to its 30,000-strong base in Europe, and attracted the interest of 50 sales representatives in the U.S. The culmination of Infogrames' North American invasion was the controlling stake in GT Interactive it purchased for $135 million in November 1999. GT Interactive had been a takeover target for some time, but it was surprising to see Infogrames manage to wrest control of GT from the Cayre family for, in effect, 75 cents a share. As we went to press, Infogrames' total sales were projected to grow by 50 percent in fiscal 2000 (the company's financial year ends in June, therefore fiscal 2000 results will be delivered at the end of summer), to reach $460 million. North American sales will contribute a fifth of this revenue.

The U.S. wasn't Infogrames' only target, however. Infogrames became a player in Australia by taking a 62.5 percent-share holding in the game distribution company Ozisoft, which had 1998 sales of 32 million Australian dollars. Then there was the acquisition of Arcadia in Spain and Portugal's A+ Multimédia, which were renamed Infogrames Spain and Infogrames Portugal respectively. These two acquisitions alone gave Infogrames direct access to more than 1,500 additional European sales outlets.

Omid Rahmat is bidding farewell to his loyal readers after this column to join Expertcity.com, based in Santa Barbara, Calif. You can still read his research and market analysis notes on his web site at http://www.smokezine.com.

Your Neighborhood Billion-Dollar Game Company

So what does consolidation buy you in today's game industry? Probably the best answer is to look at what Gremlin, Accolade, and GT Interactive bring to the table. Obviously, all these companies generate revenue from the sale of games. However, at least in the case of Gremlin and GT Interactive, both public companies in their respective countries, the combined impact of a share-price collapse and a lack of capital made them easy prey. However, the real key to assessing the value of these companies is in examining the mix of development talent, franchise titles, and distribution that they offer.

Gremlin's U.K. sales account for 56 percent of its business, with 25 percent in Europe and 19 percent for the rest of the world. However, Gremlin also has a development business. The company acquired Scottish developer DMA Design in 1996, and 80 percent of its titles are developed in-house. Among Gremlin's titles are ACTUA SOCCER (46 percent of 1996 sales) and ACTUA GOLF products. DMA Design developed the hit LEMMINGS series, BODY HARVEST for the N64, and GRAND THEFT AUTO for Take Two Interactive. For its fiscal year ending in June 1999, Gremlin had planned 14 titles representing 20 SKUs. For all this, Infogrames paid £24 million ($39 million). Not a bad deal for a very talented development group and some interesting franchises.

A good soccer franchise is as essential to a European game developer as a football title is to a U.S. company. Therefore, in acquiring Gremlin, Infogrames adds a couple of key sports franchises as well as a strong group of developers. It also helps solidify the position of both Gremlin and Infogrames in Europe, and allows Infogrames to expand the reach of Gremlin titles beyond its home territories.

Accolade is an even more interesting acquisition. The company was founded in 1984 by former Activision developer Alan Miller. Accolade was primarily involved in the development of console titles (having made the transition from PC titles in 1991), and owns the reputable and successful HARDBALL series franchise, in addition to TEST DRIVE and TEST DRIVE OFF-ROAD, the number-one U.S. off-road franchise for the Playstation. But here's the real kick for its French parent: Accolade also has prestige licenses such as Major League Baseball, the Major League Baseball Players' Association, and automotive licenses such as Jaguar, Hummer, Land Rover, Jeep Wrangler, Chevrolet Corvette, Chevrolet Camaro, Dodge Viper, Dodge T-Rex, Dodge Ram, and Shelby Cobra. It's a product base of testosterone and pure Americana that is not exactly in line with Infogrames' traditional image.

It has to be said that European technology companies traditionally find it difficult to operate successfully in the North American market. It may be partly cultural, and it may also be partly the competitive nature of the U.S. market coupled with its complex mix of channels and tastes. Into this fray, Infogrames has added its 70 percent stake in GT Interactive, indicating that the company is confident in its ability to handle its U.S. outlets, but that's where its greatest challenge lies.

	Financial Year 1997-98	Financial Year 1998-99	% Change
Sales	$230.8	$316.1	+37%
Gross Profit	97.1	140.5	+44.7
R&D	(19.8)	(26.6)	+34.4
Marketing & Sales	(33.4)	(57.6)	+72.6
Fixed Costs	(27.9)	(29.4)	+ 5.3
Operating Income	16.0	26.8	+67.5
Income before Taxes & Goodwill	12.4	27.9	+124.9
Consolidated Net Income	13.3	21.3	+60.2

Infogrames' financials for 1997–98 and 1998–99, expressed in millions of dollars based upon an exchange rate of 1 Euro = US$1.03280. Infogrames' fiscal year ends in June, and its acquisitions during 1999 did not contribute significantly to the results of the financial year ending June 30, 1999. For instance, Accolade and Gremlin were consolidated on June 1 and June 30, 1999, respectively.

The Global Imperative

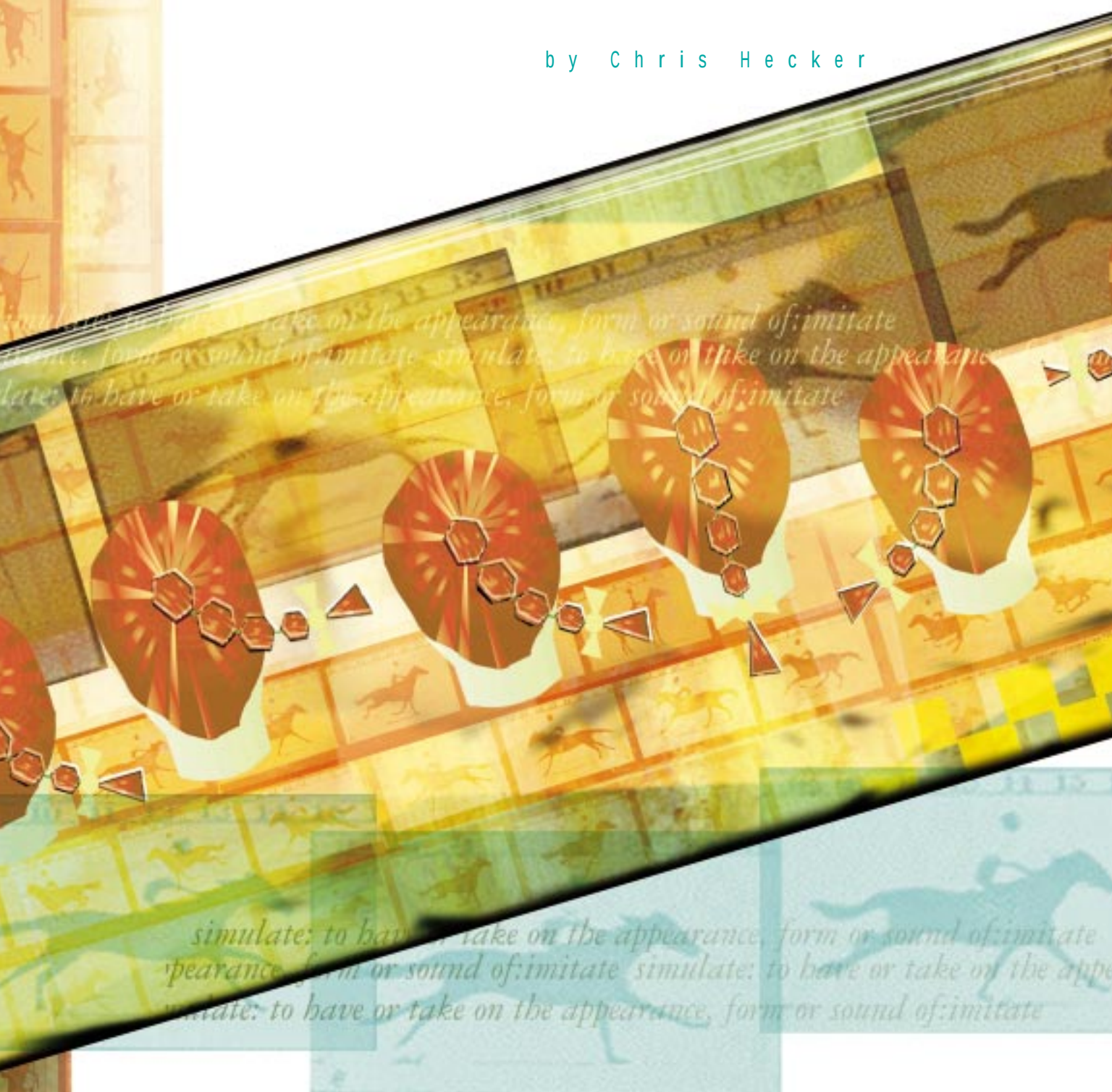
Whatever the value of Infogrames' acquisitions and however the resultant mix of companies performs, I think Infogrames makes an excellent case study in how to position yourself as a game company in the next five years — you have to be global. Gremlin, Accolade, and GT Interactive can benefit from Infogrames' European presence as much as Infogrames benefits from its increased presence in English-speaking markets. I'd even go as far as to say that almost all the major European and American game companies are starting to resemble Electronic Arts, the company that helped define what it means to have a global brand in this business.

Still, there are numerous problems in Infogrames' strategy. GT Interactive could end up being a drain. GT isn't a pure game company; it also has educational and reference products. It has great distribution in North America, but could nevertheless end up being a big weight on Infogrames' shoulders. Infogrames has a ways to go in creating a unique identity and family of franchises to accompany its global presence. The company has some consolidating to do on its product lines, its development slate, and even in its distribution channels. It's a common side effect of rapid growth. Still, had Infogrames, Ubi Soft, Titus, and Eidos not been so aggressive in the past three years, the European game industry would have been in danger of becoming swamped by U.S. takeovers, or at worst being marginalized as a development resource for American and Japanese companies. ■

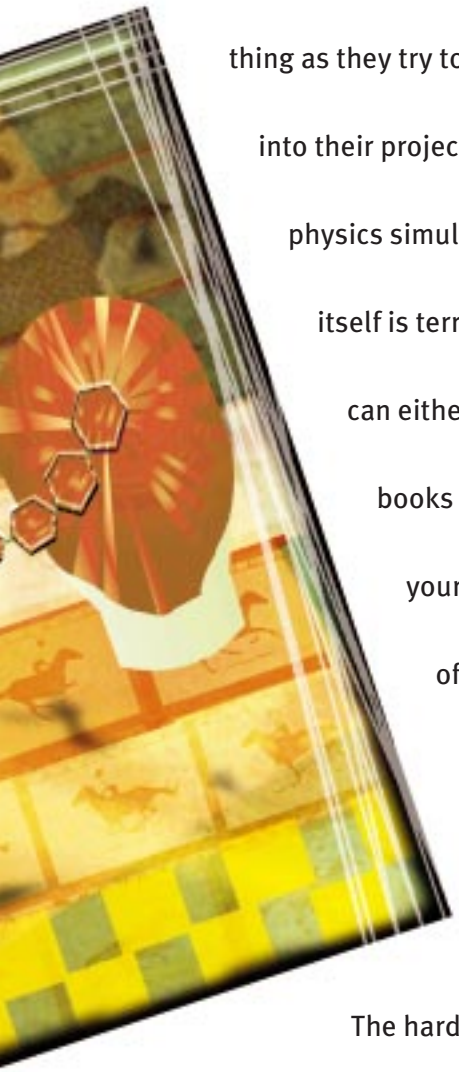
Back

How to Simulate a Ponytail

by Chris Hecker



The truth is, it's hard to use physics in games. I found this out the hard way, and I think a lot of other developers are finding the same thing as they try to integrate dynamics into their projects. It's not that the physics simulation technology itself is terribly difficult — you can either read a bunch of books and implement it yourself, or license one of the many physics simulators on the market these days.



The hard part is integrating dynamics with game play in a meaningful way.

Always one to avoid the hard parts when possible, I'm going to present a slightly different kind of physics article this time around. Let's completely dodge the integration of physics and game play, and simply use physics to dynamically generate some cool effects that would be tedious or difficult for an animator to create by hand. This isn't a cop out or a totally superfluous goal, mind you. Not only are special effects important to games as we all know, but by easing physics into the development process

Chris Hecker (checker@d6.com) can't come up with a funny saying for his bio because he's been highly constrained by the deadline.

through relatively low-risk special effects, you can get comfortable with the math and implementation in a real, shipping game. The experience gained through incremental adoption will be very valuable when you're deciding whether to add physics to any of the core game-play elements in your game, and possibly risking the project in the process. To this end, we're going to simulate a character's hair tied in a ponytail.

The Ponytail

No one would argue that the ponytail is the most important physical feature of today's game heroines (ahem), but ponytails have a lot of characteristics that make them compelling candidates for simulation. First and foremost — given our focus on low-risk special effects in this article — ponytails are relatively important to the look of a character, but they don't affect the game play. Rarely are videogames won or lost based on the movement of a ponytail. Second, the ponytail's movement is almost always passively dynamic, depending only on external forces like gravity and the character's movement. Games don't often need the ponytail to move in a specific way, it just has to look like a ponytail. This is the best kind of animation to simulate, because not only is it the easiest (as opposed to simulating something with active controlled dynamics, such as a creature's walking motion), but it's also the most tedious to hand-animate. If we can write a short piece of code to dynamically generate the ponytail motion given any possible movement of the character and the forces acting on her, then our animator can go work on something more important. We get a ponytail that always reacts correctly, rather than having only a few canned ponytail animations. Finally, the math behind the ponytail simulation we'll derive is applicable to any other dynamic chain, and this category includes all sorts of other objects you see in games, such as ropes and chains hanging from ceilings, swords in scabbards on characters' belts, and the like.

FIGURE 1. *A screenshot from the sample application showing a ponytail swinging from the back of a head.*

There are myriad ways to simulate a ponytail, both in the sense that there are a large number of physical models for a ponytail and a large number of ways to simulate each model. Picking an appropriate model for your system that captures the dynamics you're interested in but doesn't make the simulation too complicated is an important first step. One obvious model for a ponytail would be to simulate every strand of hair and actually tie the simulated strands together into a conventional ponytail. This is probably overkill for the kind of movement we'd like to capture, not to mention that a highly accurately simulated ponytail like this would probably come undone in the middle of some Egyptian crypt, which is the last thing you want to have happen while adventuring. My own ponytail comes undone while I'm typing articles, let alone while battling lions and tigers and bears…anyway, you get the picture.

We're going to model our ponytail as a series of rigid bodies constrained together with joints. Our joints allow the bodies to rotate relative to each other, but not to translate relative to each other. Thus, the ponytail can flop around, but it can't stretch or slide apart. See Figure 1 for a screenshot of the sample application showing the ponytail. Each segment of the ponytail will be a rigid body, and each body will be attached to its two neighbors. The first and last links are exceptions. The last link is attached to the rest of the ponytail above it, but it doesn't have a neighbor below it, so it's the end of the chain and it dangles freely.

The first link is attached to its neighbor below it on the ponytail, and to the head. The ponytail bodies move dynamically due to the simulation, assuming we do our job correctly, but the head is a different matter.

Kinematic and Dynamic Control

We definitely don't want the head to move dynamically, since that opens the can of game play worms we're trying to avoid by simulating something "trivial," like a ponytail. The artist should have complete control over the head's movement, and that animation should be played back by the game exactly as if there were no simulated ponytail. So, how do we connect the dynamically simulated ponytail to the traditionally animated head? The exact mathematics for this connection will have to wait until later in the derivation, but the concept is important to discuss early on.

As you'll remember from my series on rigid body dynamics in *Game Developer* (Oct./Nov. 1996–Feb./Mar. 1997 and June 1997), the quantities we use during simulation break down into kinematic quantities and dynamic quantities (the articles are available on my web site or on the 1999 *Game Developer* back issues CD-ROM; see the end of this article for details). The kinematic quantities, such as position, velocity, and acceleration, describe the movement of the object, but don't specify why these quantities might change. The dynamic quantities, including force and mass, describe why and how the kinematic quantities are changing.

This is true for a dynamically simulated rigid body, but what about the character's head? It has animations generated by an artist in a tool such as Maya, or out of procedural animation code, not from our dynamic simulation. This kind of body is "kinematically controlled," as opposed to the dynamically controlled bodies that we've simulated before. It is kinematically controlled because there are prescribed functions for the body's kinematic quantities, whether simple interpolated keyframes for the posi-

tion, or something more elaborate. Mixing kinematically controlled bodies with dynamically controlled and simulated bodies is an important part of incrementally adopting physics for things such as special effects. We need our dynamically simulated bodies to react to the kinematically controlled bodies, but not vice versa — we always want to respect the artist's kinematic animations and leave them in control.

Our constrained rigid body model for the ponytail is obviously a simplification, but it is detailed enough to capture most of the important dynamics of the ponytail's movement. It's not modeling the flexibility of the hair except at the joints, but then again, most of the animators doing ponytails aren't doing more than linked segments anyway. Our model closely matches the bones-based animation models that most animation tools are using today.

Lagrange Multipliers

Now that we've chosen our basic physical model, we need to choose a solution method. There are a number of different techniques for simulating constrained rigid bodies, and we don't have room to discuss them even briefly here. I've chosen a popular method that's relatively intuitive and easy to implement. Perhaps most importantly, it has a mathematical derivation that fits in a magazine article or two.

The technique we're going to use is called Lagrange Multipliers. The basic idea behind this method is first to calculate the external forces and torques on the constrained rigid bodies, completely ignoring the constraints. Then we calculate the forces of constraint that keep the joints together given these external forces trying to pull the joints apart. So, in Figure 2, if Body B is pulled up by some force, we'll calculate a joint force that will pull up Body A and the constraint will stay satisfied.

The tricky part is how to calculate that joint force. Calculating this force is tricky because it depends on the dynamics of the objects. Obviously, if Body B and Body A are both traveling at the exact same velocity in the same direction, then the joint won't need to exert any force to stay together. Similarly, the joint shouldn't counteract any rotational movement, so if Body A is rotating around the joint but the position of the joint is not moving, there should be no joint force as well. Only if the joint threatens to translate apart will the algorithm compute and apply a nonzero force.

The derivation in this article is going to follow the derivation I gave in my lecture of the same name at the recent Game Developers Conference RoadTrips. As I did for that lecture, I'm going to have to assume you've either read my physics articles or their equivalents from other sources. We're going to start manipulating the dynamics equations straightaway, so go review now if you need to by using the references at the end of this article. I've placed the basic kinematic and dynamic equations for a 3D rigid body in Table 1 for quick reference.

TABLE 1. *Kinematic and dynamic equations for a 3D rigid body.*

KINEMATIC EQUATIONS. R is the position of the center of mass, r is some radius vector to a point p fixed in the object.

$$p = R + r \qquad \text{Eq. 1}$$

$$\dot{p} = \dot{R} + \omega \times r \qquad \text{Eq. 2}$$

$$\ddot{p} = \ddot{R} + \alpha \times r + \omega \times \omega \times r \qquad \text{Eq. 3}$$

DYNAMIC EQUATIONS. Equations 4 and 5 are $f=ma$ for a 3D body. Equations 6 and 7 describe how a force at p affects the center of mass.

$$f = m\ddot{R} \qquad \text{Eq. 4}$$

$$\tau = I\alpha + \omega \times L \qquad \text{Eq. 5}$$

$$f_{cm} = f_p \qquad \text{Eq. 6}$$

$$\tau_{cm} = r \times f_p \qquad \text{Eq. 7}$$

The Derivation

We'll do most of our derivation work using only two bodies with a single constraint between them. This will help us get comfortable with the math and detect the structure within it without needing to mess with lots of bodies and constraints from the beginning. Let's start by outlining the steps in the derivation:

1. Figure out notation and conventions.
2. Write dynamics equations with unknown constraint force.
3. Write constraint equation in terms of body accelerations.
4. Plug 'n' chug to get constraint equation with unknown constraint force.
5. Numerically solve for constraint force.

Step 1 is incredibly important. If you don't have your conventions worked out before you start, you'll quickly get lost in a sea of conflicting symbols. Our notations and conventions are illustrated in Figure 2. I've labeled the bodies A and B, and all objects fixed on the bodies are subscripted appropriately. So, p_A is the tip of A's constraint vector, computed by adding body A's center of mass position, R_A, to A's joint vector, r_A. Our goal is to enforce the constraint that p_A is equal to p_B in world space at all times. That is, the bodies can move around the world and rotate and whatnot, but the ends of their joint vectors had better match up or that means the joint came apart and we screwed up.

We're going to use f_c to denote the constraint force vector that's applied at the joint to keep it together. This is the vector we're trying to calculate. Although both bodies on either side of the joint feel a constraint force, there's only one constraint force per joint because of Newton's third law. This law states that for every action there's an equal and opposite reaction, or put in plain terms, whenever the joint pulls

on Body A, it pulls on Body B in exactly the opposite direction. If Body A feels the joint pulling it up, then Body B feels the joint pulling it down. This means we only need to calculate a single vector for each constraint, and then we can apply it positively to one of the bodies and negatively to the other. By convention, we will apply the constraint force positively to Body A.

The constraint force vector is a 3D vector, as is every other force in our dynamic system, including springs, drag forces, friction, and so on. At the end of all our equation manipulation, we're going to end up with a matrix equation that looks like $Af_c = b$, where A is a three-by-three matrix, and f_c and b are three-vectors. A and b will be known to us (they'll be composed of various known vectors in the system at the given time, like the positions and velocities of the objects). We will need to calculate f_c from this linear system of equations. We'll talk more about solving this system when we come to it, but while we're in the thick of things, remember our goal, $Af_c = b$. Keeping our eyes on the prize will help us stay sane and guide us in our manipulations when we're awash in equations.

The Dynamics Equations

Let's quickly write down the linear and rotational dynamics equations for Body A:

$$f_c + F_{EA} = M_A \ddot{R}_A \qquad \text{Eq. 8}$$

$$r_A \times f_c + \tau_{EA} = I_A \alpha_A + \omega_A \times L_A \qquad \text{Eq. 9}$$

I've separated out the forces and torques into those caused by f_c and those caused by the external stimuli. The latter are labeled with a subscript *E* for "external." External forces are basically "everything else," such as springs, friction, drag, forces from explosions, weapon recoil, wind blowing, and every other force and torque that affects the bodies in the system. These should all be known at the current instant. If we weren't going to simulate the constraint, we'd just plug in the external forces and all the other known terms (the masses, inertia tensors, angular velocity and momentum, and so on) and integrate forward, just like when we were simulating discrete rigid bodies. However, the unknown f_c keeps us from integrating yet, because we need to know all the forces on the objects to find the accelerations.

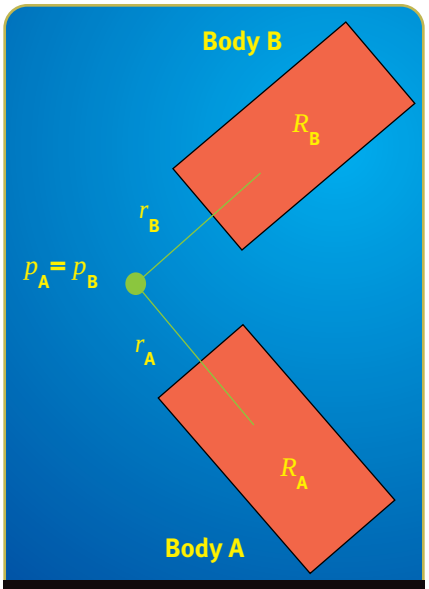


FIGURE 2. *A simple system with two bodies and one constraint between them.*

We can take Equations 8 and 9 and solve them for the linear and angular acceleration terms:

$$\ddot{R}_A = M_A^{-1} f_c + M_A^{-1} F_{EA} \qquad \text{Eq. 10}$$

$$\alpha_A = I_A^{-1}(r_A \times f_c) + I_A^{-1}\tau_{EA} - I_A^{-1}(\omega_A \times L_A) \qquad \text{Eq. 11}$$

In Equations 8 and 10 I'm treating the rigid body mass, *M*, as a matrix rather than as the usual scalar. This is totally acceptable, as long as I make this mass matrix mathematically equivalent to the scalar. It's easy to create such a matrix by multiplying the identity matrix by the mass.

Equations 10 and 11 for Body B are almost identical. Obviously we need to change the little A subscripts to little B subscripts. Besides that, the only real difference is that the constraint force is applied negatively to Body B, so wherever f_c appears in the equations for Body A, $-f_c$ will appear in those for Body B.

The Constraint Equation

Now we have four vector equations for the accelerations of the bodies: Equations 10 and 11 and their equivalents for Body B. If we knew the force of constraint *a priori*, we could plug it in here with the other known values and then compute the new accelerations of the objects and step forward in time. Since we don't know f_c yet, we need another equation to play around with. The constraint equation should do nicely.

You should notice that we haven't really talked mathematically about the constraint yet. We've said we're going to enforce a constraint, but how is that expressed in symbols? It's relatively simple. Just write an equation that describes the desired situation. I propose this:

$$p_A - p_B = 0 \qquad \text{Eq. 12}$$

Or, written out in terms of the individual body components:

$$R_A + r_A - R_B - r_B = 0 \qquad \text{Eq. 13}$$

Equation 12 (and 13) states that the vector to the endpoints of the constraints on the two bodies have to be equal. If Body A's constraint endpoint moves to the left, then Body B's had better follow or Equation 12 will be violated. If we can enforce Equation 12 at all times, we've constrained the bodies together.

It's not at all clear how to keep Equation 12 satisfied using our force, f_c, though. Forces can't directly affect positions, so we need to put Equation 12 into a form where our f_c can act upon it. The secret is to differentiate the equation twice. This will give us a constraint equation in terms of accelerations, which we know from $f = ma$ are directly influenced by forces. More specifically, differentiating Equation 12 will give us a constraint equation in terms of the bodies' accelerations, which are directly influenced by f_c via Equations 10 and 11.

Differentiating Equation 12 isn't just a symbolic trick to make it work with forces, it actually makes intuitive sense as well. Since Equation 12 says the positions of the two points must coincide, its first derivative says their velocity vectors must be equal as well. This is symbolically obvious from simply taking the derivative:

$$\dot{p}_A - \dot{p}_B = 0$$

But, it's also physically obvious when you think about it. If the joint endpoint velocities were not equal at some point in time, then an instant later their positions would have to be

unequal as well, since differing velocities means the points were going in different directions. Does this mean we need to enforce the velocity constraint as well as the positional constraint? No. If the objects begin the simulation with the position constraint satisfied, then as long as we satisfy the velocity constraint every timestep, the position equation will be satisfied automatically. How could it not be? It's satisfied at time = 0, and then we enforce the velocities to be the same at all times, so the positions can never diverge. (For extra credit, think about how this phenomenon is related to the somewhat mysterious constant that always appeared when integrating equations in high school calculus.)

This argument makes sense for acceleration as well. If the position and velocity constraints were met at some time in the past, and we've forced the accelerations of the points to be equal at all times since then, the positions must still be equal because the velocities must have always been equal. Again, we can write out the second derivative of the constraint equation and see this symbolically:

$$\ddot{p}_A - \ddot{p}_B = 0 \qquad \text{Eq. 14}$$

Now that we've arrived at the acceleration version of the constraint equation, how do we use f_c to enforce it?

For starters, Equation 14 is a pretty compact and abstract way of describing the relationship between the two joint endpoint accelerations. We can drill down and enlarge it considerably by substituting in the definition of the points' accelerations in terms of their center of mass and angular accelerations from Equation 3. If we substitute in Equation 3 for both p_A and p_B into Equation 14, we get a much more detailed description of what's going on, and we also get a big huge mess of terms.

Homework

Unfortunately, it's a mess of terms that you're going to have to battle with yourself until next month, because I'm out of space. I will drop a few hints for the adventurous. The main idea is to take the big mess we just made, and make it even bigger by substituting in Equations 10 and 11 and their equivalents for Body B into the body acceleration terms. This will give us one huge equation that we can eventually get into our goal form, $Af_c = b$. Still, you might go insane manipulating all of those terms, so if you're going to try it, I recommend only dealing with one of the joint endpoints and seeing where you can get with that first.

Next month we'll finish up the derivation for two bodies with one constraint, and talk about extending the math to arbitrary numbers of bodies and constraints. ■

FOR FURTHER INFO

My Dynamics Page
http://www.d6.com/users/checker/dynamics.htm

The 1999 Game Developer Back Issues CD-ROM
http://www.gdmag.com

Building an Advanced Particle System

by John van der Burg

Imagine a scene in a game in which a rocket flies through the air, leaving a smoke trail behind it. Suddenly the rocket explodes, sparks flying everywhere. Out of the disintegrating rocket a creature is jettisoned towards you, its body parts exploding and blood flying through the air, leaving messy blood splatters on the camera lens. What do most of the elements in this scene have in common?

Yes, most of these elements are violent. But in terms of technology, most of the effects in a scene like this would benefit from a good particle system. Smoke, sparks, and blood are routinely created in today's games using particle systems.

To realize these effects, you need to build a particle system, and not just a simple one. You need an advanced particle system, one that's fast, flexible, and extensible. If you are new to particle systems, I recommend you begin by reading Jeff Lander's article on particle systems ("The Ocean Spray in Your Face," Graphic Content, July 1998). The difference between Lander's column and this article is that the former describes the basics of particles, whereas I will demonstrate how to build a more advanced system. With this article I will include the full source code for an advanced particle system, and you can download an application that demonstrates the system.

Performance and Requirements

Advanced particle systems can result in pretty large amounts of code, so it's important to design your data structures well. Also be aware of the fact that particle systems can decrease the frame rate significantly if not constructed properly, and most performance hits are due to memory management problems caused by the particle system.

When designing a particle system, one of the first things to keep in mind is that particle systems greatly increase the number of visible polygons per frame. Each particle probably needs four vertices and two triangles. Thus, with 2,000 visible snowflake particles in a scene, we're adding 4,000 visible triangles for the snow alone. And since most particles move, we can't precalculate the vertex buffer, so the vertex buffers will probably need to be changed every frame.

The trick is to perform as few memory operations (allocations and releases)

John van der Burg is the lead programmer for Mystic Game Development, located in the Netherlands. Currently he is working on Oxygen3D, which is his third hardware-only engine and his eighth engine overall. Currently he is doing freelance work on LOOSE CANNON for Digital Anvil and for OMG Games on THE CREST OF DHARIM. He previously freelanced for Lionhead Studios on BLACK AND WHITE, and for Orange Games on CORE. You can find screenshots of his previous work at http://www.mysticgd.com. Feel free to drop him a line at john@mysticgd.com.

as possible. Thus, if a particle dies after some period of time, don't release it from memory. Instead, set a flag that marks it as dead or respawn (reinitialize) it. Then when all particles are tagged as "dead," release the entire particle system (including all particles within this system), or if it's a constant system, keep the particle system alive. If you want to respawn the system or just add a new particle to a system, you should automatically initialize the particle with its default settings/properties set up according to the system to which it belongs.

For example, let's say you have a smoke system. When you create or respawn a particle, you might have to set its values as described in Table 1. (Of course, the start color, energy, size, and velocity will be different for blood than, say, smoke.) Note that the values also depend on the settings of the system itself. If you set up wind for a smoke system so the smoke blows to the left, the velocity for a new particle will differ from a smoke system in which the smoke just rises unaffected by wind. If you have a constant smoke system, and a smoke particle's energy becomes 0 (so you can't see it anymore), you'll want to respawn its settings so it will be replaced at the bottom of the smoke system at full energy.

Some particle systems may need to have their particles rendered in different ways. For example, you may want to have multiple blood systems, such "blood squirt," "blood splat," "blood pool," and "blood splat on camera lens," each containing the appropriate particles. "Blood squirt" would render blood squirts flying through the air, and when these squirts collided with a wall, the "blood splat" system would be called, creating messy blood splats on walls and floors. A blood pool system would create pools of blood on the floor after someone had been shot dead on the ground.

Each particle system behaves in a unique manner. Blood splats are rendered differently than smoke is displayed. Smoke particles always face the active camera, whereas blood splats are mapped (and maybe clipped) onto the plane of the polygon that the splat collides with.

When creating a particle system, it is important to consider all of the possible parameters that you may want to affect in the system at any time in the game, and build that flexibility into your system. Consider a smoke system again. We might want to change the wind direction vector so that a car moving closely past a smoke system makes the smoke particles respond to the wind generated by the passing car.

At this point you may have realized that each of these systems (blood splat, smoke, sparks, and so on) is very specific to certain tasks. But what if we want to control the particles within a system in a way not supported by the formulae in the system? To support that kind of flexibility, we need to create a "manual" particle system as well, one that allows us to update all particle attributes every frame.

The last feature we might consider is the ability to link particle systems within the hierarchy of our engine. Perhaps at some point we'll want to link a smoke or glow particle system to a cigarette, which in turn is linked to the head of a smoking character. If the character moves its head or starts to walk, the position of the particle systems which are linked to the cigarette should also be updated correctly.

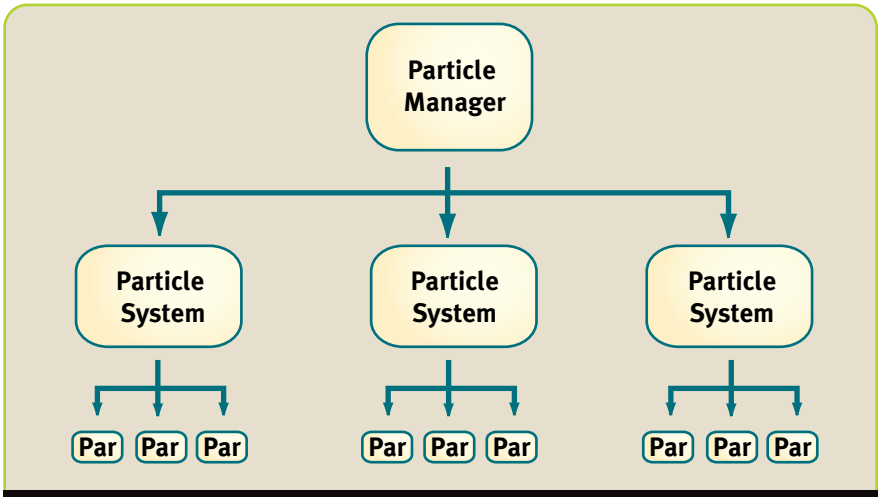


FIGURE 1. *A global overview of the system we are going to build.*

TABLE 1. *Particle attributes.*

Data type	Name	Description
Vector3	position	The position of the particle in world-space
Vector3	oldPos	The previous position of the particle, useful in some systems
Vector3	velocity	The velocity vector (position += velocity)
dword	color	The color of the particle (its vertex colors)
int	energy	The energy of the particle
float	size	The size of the particle

So there you have some basic requirements for an advanced particle system. In the next section, I'll show how to design a good data structure that is capable of doing all the above-mentioned features.

Creating the Data Structure

Now that we know what features we need, it's time to design our classes. Figure 1 shows an overview of the system we're going to build. Notice that there is a particle manager, which I will explain more about in a moment.

Let's use a bottom-up approach to design our classes, beginning with the particle class.

THE PARTICLE CLASS. If you have built a particle system before, you probably know the types of attributes a particle must have. Table 1 lists of some common attributes.

Note that the previous position of a particle can also be useful in some systems. For example, you might want to stretch a particle between its previous and current positions. Sparks are a good example of particles that benefit

FIGURE 2. *Some spark effects you can create using a particle system such as the one discussed in this article. Note that all particles perform accurate collision detection and response in these two screenshots.*

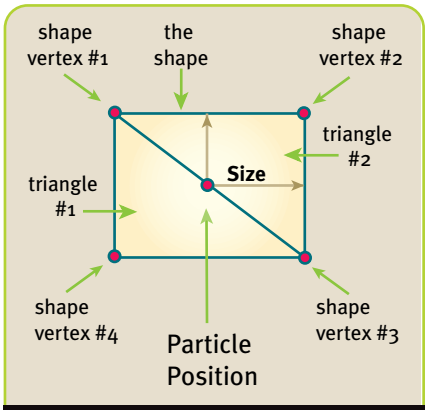


FIGURE 3. *Setting up a shape to render a particle.*

LISTING 1. *Calculating the shape of a particle facing the camera.*

```
void ParticleSystem::SetupShape(int nr)
{
    assert(nr < shapes.Length());     // make sure we don't try to shape anything we
                                      // don't have

    // calculate cameraspace position
    Vector3 csPos = gCamera->GetViewMatrix() * particles[nr].position;

    // set up shape vertex positions
    shapes[nr].vertex[0] = csPos + Vector3(-particles[nr].size, particles[nr].size, 0);
    shapes[nr].vertex[1] = csPos + Vector3( particles[nr].size, particles[nr].size, 0);
    shapes[nr].vertex[2] = csPos + Vector3( particles[nr].size, -particles[nr].size, 0);
    shapes[nr].vertex[3] = csPos + Vector3(-particles[nr].size, -particles[nr].size, 0);
}
```

from this feature. You can see some spark effects I've created in Figure 2.

The color and energy attributes can be used to create some interesting effects as well. In a previous particle system I created, I used color within the smoke system, which let me dynamically light the smoke particles using lights within the scene.

Energy value is very important as well. Energy is analogous to the age of the particle — you can use this to determine whether a particle has died. And because the color or intensity of some particles (such as sparks) changes over time, you may want to link it to the alpha channel of the vertex colors.

I strongly recommend that you leave the constructor of your particle class empty, because you don't want to use default values at construction time, simply because these values will be different for most particle systems.

THE PARTICLE SYSTEM CLASS. This class is the heart of the system. Updating the particle attributes and setting up the shape of the particles takes place inside this class. My current particle system class uses the node base class of my 3D engine, which contains data such as a position vector, a rotation quaternion, and scale values. Because I inherit all members of this node class, I can link my particle systems within the hierarchy of the engine, allowing the engine to affect the position of the particle system as discussed in the above cigarette example. If your engine does not have hierarchy support, or if you are building a stand-alone particle system, this is not needed. Table 2 lists the attributes which you need to have in the particle system base class.

Here's how to calculate the four positions of a normal (not stretched) particle that always faces the active camera. First, transform your particle world-space position into camera-space (multiply the world-space position and your active camera matrix) using the size attribute of the particle to calculate the four vertices.

The four vertices, which form the shape, are what we use to render the particle, though a particle has only one position, *xyz*. In order to render a particle (such as a spark), we need to set up a shape (created from four vertices). Two triangles are then rendered between these four points. Imagine a non-stretched particle always facing the camera in front of you, as seen in Figure 3. In our case, the particle is always facing the active camera, so this means we can simply add and subtract values from the *x* and *y* values of the particle position in camera-space. In other words, leave the *z* value as it is and pretend you are working only in 2D. You can see an example of this calculation in Listing 1.

THE FUNCTIONS. Now that we know what attributes are needed in the particle system base class, we can start thinking about what functions are needed. Since this is the base class, most functions are declared as virtual functions. Each type of particle system updates particle attributes in a different way, so we need to have a virtual update function. This update function performs the following tasks:

- Updates all particle positions and other attributes.
- Updates the bounding box if we can't precalculate it.
- Counts the number of alive particles. It returns FALSE if there are no alive particles, and returns TRUE if particles are still alive. The return value can be used to determine whether a system is ready to be deleted or not.

TABLE 2. *Particle system base class attributes.*

Data type	Name	Description
Texture	*texture	A pointer to a texture, which all particles will use. For performance reasons, we only use one texture for each individual particle system; all particles within the specific system will have the same texture assigned.
BlendMode	blendMode	The blend mode you want to use for the particles. Smoke will probably have a different blend mode from blood — that's the reason you also store the blend mode for each particle system.
int	systemType	A unique ID, which represents the type of system (smoke or sparks, for example). The systemType identifier is also required, since you may want to check for a specific type of particle system within the collection of all systems. For example, to remove all smoke systems, you need to know whether a given system is a smoke system or not.
Array Particle	particles	The collection of particles within this system. This may also be a linked list instead of an array.
Array PShape	shapes	A collection of shapes, describing the shapes of the particles. The shape descriptions of the particles usually consist of four positions in 3D camera-space. These four positions are used to draw the two triangles for our particle. As you can see in Table 1, a particle is only stored as a single position, but it requires four positions (vertices) to draw the texture-mapped shape of the particle.
int	nrAlive	Number of particles in the system which are still alive. If this value is zero, it means all particles are dead and the system can be removed.
BoundingBox3	boundingBox	The 3D axis-aligned bounding box (AABB), used for visibility determination. We can use this for frustum, portal, and anti-portal checks.

Now our base class has the ability to update the particles, and we are ready to set up the shapes which can be constructed using the new (and perhaps previous) position. This function, SetupShape, needs to be virtual, because some particle system types will need to have their particles stretched and some won't. You can see an example of this function in Listing 1.

To add a particle to a given system, or to respawn it, it's useful to have a function that takes care of this. Again, it should be another virtual function, which is declared like this:

```
virtual void SetParticleDefaults( Particle &p );
```

As I explained above, this function initializes the attributes for a given particle. But what if you want to alter the speed of the smoke or change the wind direction that affects all of your smoke systems? This brings us to the next subject: the particle system's constructor. Many particle systems will need their own unique constructors, forcing us to create a virtual constructor and destructor within the base class. In the constructor of the base class, you should enter the following information:

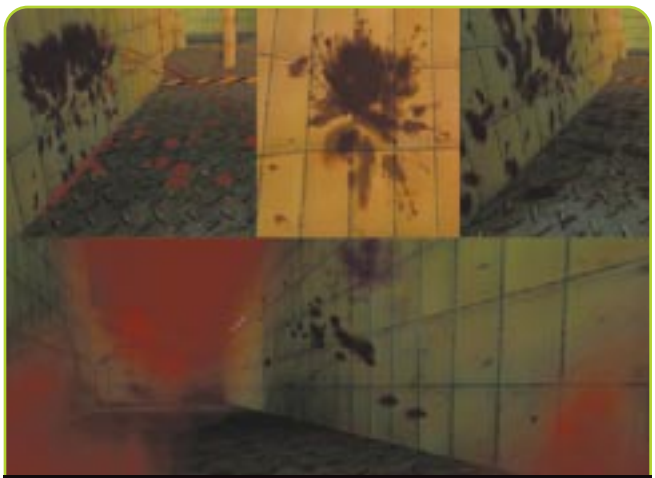
- The number of particles you initially want to have in this particle system.
- The position of the particle system itself.
- The blend mode you want to use for this system.
- The texture or texture file name you want this system to use.
- The system type (its ID).

In my engine, the constructor in the particle system base class looks like this:

```
virtual ParticleSystem(int nr, rcVector3 centerPos, BlendMode
blend=Blend_AddAlpha, rcString file name="Effects/Particles/
green_particle", ParticleSystemType type=PS_Manual);
```

So where do various settings, such as the wind direction for the smoke system, get addressed? You can either add settings specific to the system type (such as wind direction) into the constructor, or you can create a struct called InitInfo inside each class, which contains all of the appropriate settings. If you use the latter method, make sure to add a new parameter in the constructor, which is a pointer to the new struct. If the pointer is NULL, the default settings are used.

As you can imagine, the first solution can result in constructors with many parameters, and that's not fun to work with as programmer. ("Parameter number 14...hmmm. What does that value represent again?") That's the main reason I don't use the first method. It's much easier to use the second



A blood system. Blood colors were set based on the colors in the light maps. Blood on dark areas looks dark as well. The red areas in the bottom screenshot are the blood splats on the camera lens, dripping down the lens.

TABLE 3. *Particle manager class functions.*

`Init`	Initializes the particle manager.
`AddSystem`	Adds a specified particle system to the manager.
`RemoveSystem`	Removes a specified particle system.
`Update`	Updates all active particle systems and removes all systems which died after the update.
`Render`	Renders all active and visible systems.
`Shutdown`	Shuts down the manager (removes all allocated systems).
`DoesExist`	Checks whether a given particle system still exists in the particle manager (if it has not been removed yet).

method, and we can create a function in each particle system class to initialize its `struct` with default settings. An example of this code and a demo application can be found on the *Game Developer* web site (http://www.gdmag.com) or my own site at http://www.mysticgd.com.

The Particle Manager

Now that we have covered the technology behind an individual particle system, it's time to create a manager class to control all of our various particle systems. A manager class is in charge of creating, releasing, updating, and rendering all of the systems. As such, one of the attributes in the manager class must be an array of pointers to particle systems. I strongly recommend that you build or use an array template, because this makes life easier.

The people who will work with the particle systems you create want to add particle systems easily. They also don't want to keep track of all the systems to see if all of the particles died so they can release them from memory. That's what the manager class is for. The manager will automatically update and render systems when needed, and remove dead systems.

When using sporadic systems (systems which die after a given time), it's useful to have a function that checks whether a system has been removed yet (for example, if it still exists within the particle manager). Imagine you create a system and store the pointer to this particle system. You access the particle system every frame by using its pointer. What happens if the system dies just before you use the pointer? Crash. That's why we need to have a function which checks if the system is still alive or has already been deleted by the particle manager. A list of functions needed inside the particle manager class is shown in Table 3.

This was constructed from sparks and a big real-time calculated flare explosion (not just a texture).

This image is the same as the above one, but with some extra animated explosions (animated textures) and shockwaves, which are admittedly very small and may be difficult to see in this image.

The AddSystem function will probably have just one parameter: the pointer to the particle system which is of the type of our particle system base class. This allows you to add a smoke or fire system easily depending on your needs. Here is an example of how I add a new particle system in my engine:

```
gParticleMgr->AddSystem( new Smoke(nr
SmokeParticles, position, ...) );
```

During the world update function, I call the gParticleMgr->Update() function, which automatically updates all of the systems and releases the dead ones. The Render function then renders all visible particle systems.

Since we don't want to keep track of all particles across all of our systems every frame to see whether all particles have died (so the system can be removed), we'll use the Update function instead. If this function returns TRUE, it means that the system is still alive; otherwise it is dead and ready to be removed. The Update function of the particle manager is shown in Listing 2.

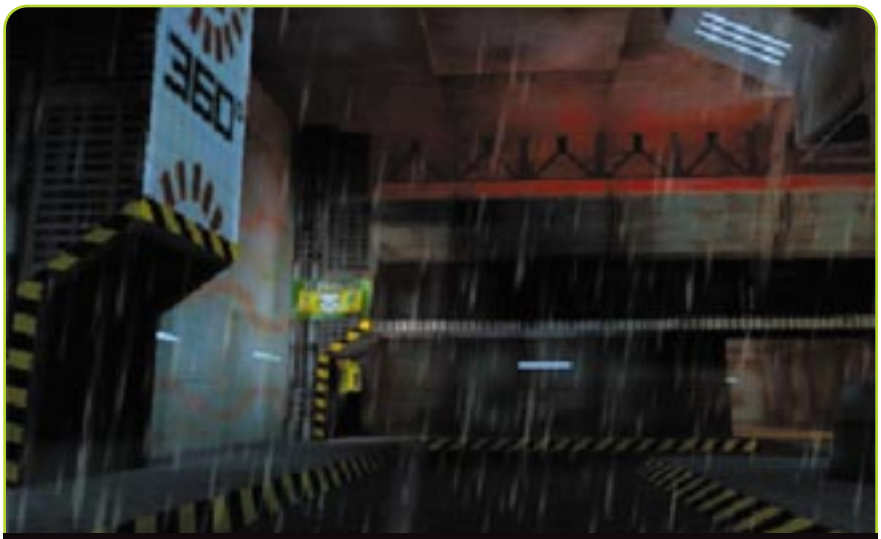
In my own particle system, all particles with the same textures and blend modes assigned to them will be rendered consecutively, minimizing the number of texture switches and uploads. Thus, if there are ten smoke systems visible on screen, only one texture switch and state change will be performed.

Design, Then Code

Designing a flexible, fast, and extensible advanced particle system is not difficult, provided you take time to consider how you will use it within your game, and you carefully design your system architecture accordingly. Because the system I discussed uses classes with inheritance, you can also put the individual particle system types into .DLL files. This opens up the possibility of creating some sort of plug-in system, which might be of interest to some game developers.

You can also download the source code of my particle system, which I have created for Oxygen3D, my latest engine. This source is not a standalone compilable system, but it should help you if you run into any troubles. If you still have any questions or remarks, don't hesitate to send me an e-mail. ■

This electricity has its own render function. A hierarchy tree was constructed to represent the electricity flow using branches and sub-branches. It is a thunderstorm lightning effect with the branches animated. Particle shapes are being constructed for every part in the electricity tree.



A rain effect, using stretched shapes for the particles. The rain also splats on the ground by calling the sparks system with adjusted settings and texture.

LISTING 2. *Update function of the particle manager.*

```
for (int i=0; i<particleSystems.Length())       // traverse all particle systems
{
     if (!particleSystems[i]->Update())         // if the system died, remove it
     {
       delete particleSystems[i];               // release it from memory
       particleSystems.SwapRemove(i);           // remove number i, and fill the gap
                                                // with the last entry in the array
     }
     else
       i++;
}
```

Surreal Software's Drakan: Order of the Flame

by Stuart Denman

The merging of great concepts from many different sources in order to create a new, better whole is perhaps one of the most fundamental aspects of human innovation. DRAKAN: ORDER OF THE FLAME uses this notion to full advantage by combining action and adventure game concepts with sword combat, aerial battles, and simple RPG elements. It is a true hybrid of many proven gaming concepts. But this attribute made DRAKAN's development doubly challenging because we had to create a game in which multiple elements worked well independently yet blended together seamlessly. Perhaps this is analogous to the way developers must work well as individuals and effectively as a team.

Stuart Denman was the lead programmer on DRAKAN and is a co-founder of Surreal Software. This was his first "real" job following four years as a student of computer engineering at the University of Washington. After scouting around Europe on a post-DRAKAN vacation, he is currently back at work developing Surreal's next 3D engine technology. He can be reached at stu@surreal.com.

Origins of the Team

Because DRAKAN was Surreal's first product, the story of DRAKAN's development is also the story of Surreal's development as a company. Surreal's creation is the classic game development story in which four ambitious recent college graduates decided they had nothing to lose and formed a game company. These four founders contributed four critical skills to the team: art, programming, design, and business skills. None of us had ever run a company or managed schedules, but we all loved games, and we knew what it took to make a good one.

Lead designer Alan Patmore had always played games and had the business savvy to complement Nick Radovich's business experience and connections. I had been programming games and graphics since the age of ten, so even though I didn't have experience working at a game development company, I did have the skills and motivation. Mike Nichols, our creative director, came from within the industry and was the only member with any titles under his belt.

Our initial goal was to develop several game concepts and a solid technological foundation that we could pitch to game publishers. This would get us the funding we needed to pay ourselves and start hiring programmers and artists without having to involve venture capitalists.

Once we got project funding, we were able to quickly build a strong team of artists, programmers, and designers who all played games. Some of the team came from other game companies — lured by the informal atmosphere and the focus on games, not profit. Others were inexperienced with game development, but had the skills and fresh ideas we needed.

As the technology lead, I was determined to build Surreal's foundations on its technology. By retaining rights to our engine and tools, we always had something to fall back on if a game design was cancelled by the publisher. This also allowed us to develop multiple game titles from one generic technology and license the technology to other companies. Any investment in time that the programmers and I put into the engine could be quickly put to use on another project if anything went awry.

We initially moved away from the popular DOOM-type engines toward a landscape-style rendering engine in order to set our games apart. There were many unique ideas that we could build from this: flying, underwater environments, outdoor deathmatch, and so on. But the technology was not only about rendering; the tools had to empower the designers and be general enough to support almost any game. So I designed a toolset in which every game-specific property and behavior would be provided by the game code itself, and the editor would be just a generic interface to the underlying game specifics.

Origins of the Beast

After pitching several game ideas to all the major publishers, we finally sold the first "dragon" concept to Virgin Interactive Entertainment (VIE) in the summer of 1996. The concept was very different from today's DRAKAN. The first concept was for a dragon RTS game in which the player's dragon could fly around taking over villages and forcing them to do their bidding. VIE wanted a more arcade-style shooter game to fill a slot in their product line, so we started developing a fast-paced, third-person dragon-flying game.

The DRAKAN team: FRONT ROW, FROM LEFT TO RIGHT: *Satish Bhatti (network programmer), Tim Ebling (programmer), Todd Andersen (designer), Susan Jessup (artist), Louise Smith (artist/animator), Andre Maguire (designer), Mel Guymon (lead animator), Tom Vykruta (programmer).*

MIDDLE ROW: *Shaun Leach (programmer), Armen Levonian (programmer), John Whitmore (designer), Greg Alt (programmer), Heron Prior (animator), Tom Byrne (artist).*

BACK ROW: *Stuart Denman (lead programmer), Scott Cummings (animator), Boyd Post (sound engineer), Alan Patmore (lead designer), Hugh Jamieson (character artist), Mike Nichols (lead artist), Hans Piwenitzky (artist), John McWilliams (designer), Nick Radovich (business/sound).*

NOT PICTURED: *Joe Olson (artist), Duncan (designer), Isaac Barry (designer), Ben Olson (artist).*

It was not until early 1997 (when VIE began cutting projects just prior to closing its doors) that Surreal sold the DRAKAN concept to Psygnosis. Psygnosis saw the strength in our team and gave us complete freedom to perfect the design. We wanted more of an RPG feel, but as a dragon, the

DRAKAN: ORDER OF THE FLAME

Surreal Software Inc.
Seattle, Wash.
(206) 587-0505
http://www.surreal.com
Release date: August 1999
Intended platform: Windows 95/98
Project budget: $2.5 million
Project length: 28 months
Team size: 23 full-time developers, 2 sound and music contractors
Critical development hardware: Pentium II and AMD K6-2 (3DNow!), 200 to 450MHz, 128MB RAM with Nvidia Riva 128 and TNT, 3dfx Voodoo 2 3D hardware. Artist workstations: Wacom tablets.
Critical development software: Windows software, Programming software: Microsoft Visual C++ 5.0 and 6.0, Visual SourceSafe 5.0, Intel VTune 2.5, InstallShield International 5.0. Art and animation software: Softimage, 3D Studio Max, Adobe Photoshop, In-house modeling and texturing tools. Sound and music: Sonic Foundry Soundforge, Emagic Logic Audio

player was limited in what he or she could carry or interact with. Adding a human rider was the best solution, and a female character was the natural choice since she would be the ideal personality to offset the dragon's immense size and power. With an increased budget under Psygnosis, we hired more team members and increased the art and game-play content to a level that the press called "ambitious" at our public debut at E3 in 1998.

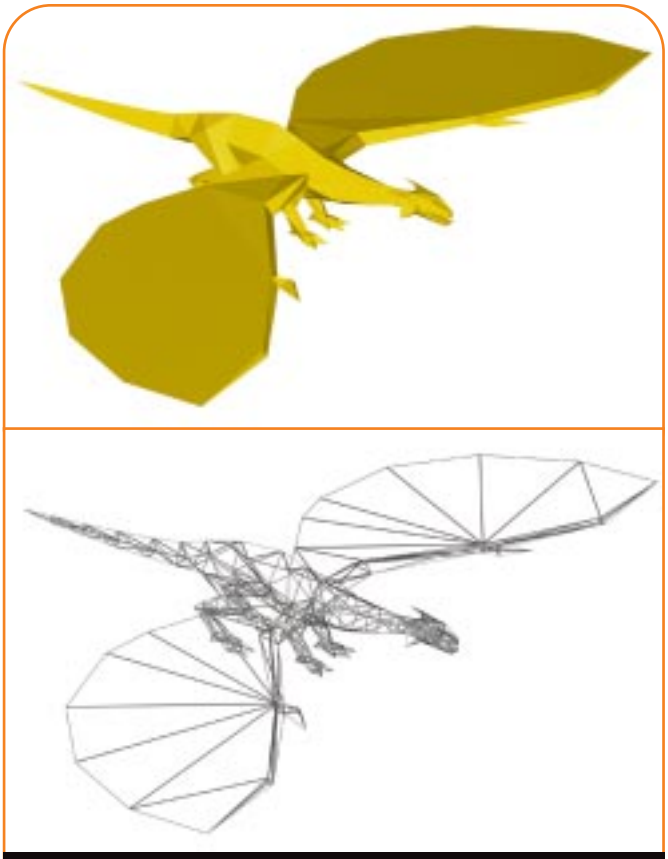
The production under Psygnosis allowed us to expand the technology as well. We added real-time lighting effects and expanded the simple height-field landscape engine into our seamless indoor/outdoor layer technology. Critical to this technology was Psygnosis's willingness to drop support for software rendering (a risky marketing decision at the time). This allowed us unprecedented freedom. We switched over to true-color textures, increased the polygon counts throughout the game, and built arbitrary geometry for our worlds. The downside to relying on 3D hardware was that we faced serious compatibility challenges — the game would have to run on almost every 3D card. This also meant battling Direct3D driver bugs, and the possibility that we would be inundated with technical support calls, since people would not have software rendering to fall back on if the 3D hardware failed to work correctly.

What Went Right

1. Success with graphics. There's no doubt DRAKAN had an ambitious design, so the graphics had to be top-notch in order to make the game world believable. The amount of art and animation content we would need mandated careful planning, lest our schedule slip. The solution to the problem was what I call "flexible reuse." In addition to the sharing of texture and geometry data between objects, DRAKAN's engine (code-named the Riot engine) was programmed to allow arbitrary scaling and rotation of art content. By assigning different behaviors and combining multiple art components, we were also able to create totally new structures with minimal effort.

Because we dropped 3D software rendering, we knew all of our textures could be created in true color. This vastly improved the look of DRAKAN, so much so that we decided to switch from using palette-based textures to true-color textures, which required quite a bit of reworking on most of the textures in the first few levels. This decision is just one example of Surreal's aesthetic fussiness. Often if a few people thought that something within the game didn't look good enough, it would end up getting redone until everyone was satisfied. The benefits can be seen in the final product, but our schedule sometimes suffered as a result.

Though the artists created the objects and buildings in the game, the designers were responsible for placing the objects into the game and gave immediate aesthetic and game-play feedback to the artists. They also were responsible for building the landscapes and caves, which defined the overall level flow. This process evened out the workload between artists and designers, but it required the designers to have a good artistic sense. This can be seen in the very fantastical landscape architectures that the designers constructed and then painted with tileable textures. The textures were drawn by the artists to have many variations and transitions, which added to the organic nature of the terrain.



Arokh's polygon mesh and alpha-blended wings (above).

2. A green team with fresh ideas. DRAKAN had an advantage that many large game development companies sometimes overlook. It had a young team, highly motivated, bursting with ideas, and ready to take risks. The ideas were unique and motivated by the desire to set DRAKAN apart from the shooters and TOMB RAIDER clones (although this was still difficult given the tendency of the gaming press to compare games to one another).

The most original idea in DRAKAN was the combination of dragon flight with sword and bow combat on the ground. This fundamental idea formed a developer's carnival for more innovative ideas and forced the player to strategize in a way not often seen in action games. The relative vulnerability of the female rider contrasted with the powerful dragon required careful thinking by the

A panoramic view of the first level in DRAKAN.

designers. Levels were created with restrictions on the dragon's ability to go places. Rynn could enter caves, but would come across areas where the dragon's flying abilities or strength would be necessary to proceed. The player (as Rynn) would then have to find a large door or other method to get the dragon inside the cave system. In this world of magic, creative ideas for special effects are very important, and these tasks were ideally suited to people who were not afraid to do things "outside the box."

3. ENGINE AND TOOLS. Many recent games have shipped with engines that simply cannot handle the target platforms and the breadth of 3D hardware that they claim to support. I believe this is primarily due to a lack of planning and preparation for current and future technologies (not having scalability, for example), and a rush to focus on the game's design and art. No time is given to solidify the underlying technology, which should ideally be done before the designers and artists even start constructing content. If the designers spend half their time waiting for the game to load, or dealing with unplayable frame rates, the final game will only be half as good as it could have been.

Regarding the game's code, if ever there was an example to put the C vs. C++ argument to rest, it is DRAKAN. There simply are no performance reasons not to go with C++, as long as the programmers understand what is happening under the covers. Object-oriented code generates so many benefits, especially for an engine that you plan to build on for many years to come. In DRAKAN, the game-specific source code and engine source code were separated into different projects, so no game-specific code was allowed in the engine. The game-specific code included such features as the user interface, AI, and game entities, and contained no platform-specific code.

The engine is broken up into many classes that handle various engine tasks and are the interfaces by which the game code accesses the engine. For instance, there is a sound class for playing sounds, a texture class for working with textures, a sequencing class for playback of scripted cutscenes, and numerous others. These also form a framework for future porting of the system-specific functions to other platforms. The stability of this system can be validated; we are currently creating additional games based on the DRAKAN engine with little or no code changes to the engine project. To further reduce the debugging time, we put coding guidelines in place to ensure the consistency of code between programmers, and created classes to catch array boundary violations and memory allocation problems.

DRAKAN has no scripting language. Instead, the programmers create modules that are visually connected by the designers to create scripted events in the game. Such modules include triggers, switches, timers, counters, and more complex modules such as doors, enemy creatures, and weapons. The modules have programmer-defined parameters associated with them. A parameter can be almost anything: a number, a list of options, a sound, a texture, another module, and so forth.

The system meant designers could tweak parameters and combine modules in ways that the programmers never intended. One particularly nice example was an effect that was originally created for the "ice sword." The effect was made up of a number of particles (originally snowflakes) that would collect for a certain amount of time on the mesh of an affected object. After a time, the particles would fall to the ground and stick for a bit. All these properties, from the timings to the particle texture, are configurable. With this feature at their disposal, the designers created glowing auras around ghosts by increasing the particle size and making the stick-time infinite. They created snow that landed on invisible platforms to guide players across them. The snow effect was attached to arrows to drop ice behind them as they flew. All this from a small bit of programming.

Surreal's in-house level editing tool showing the real-time 3D editing window in the center and top-down layout view in the upper-left.

The engine also has an efficient caching system, so it's able to handle hun-

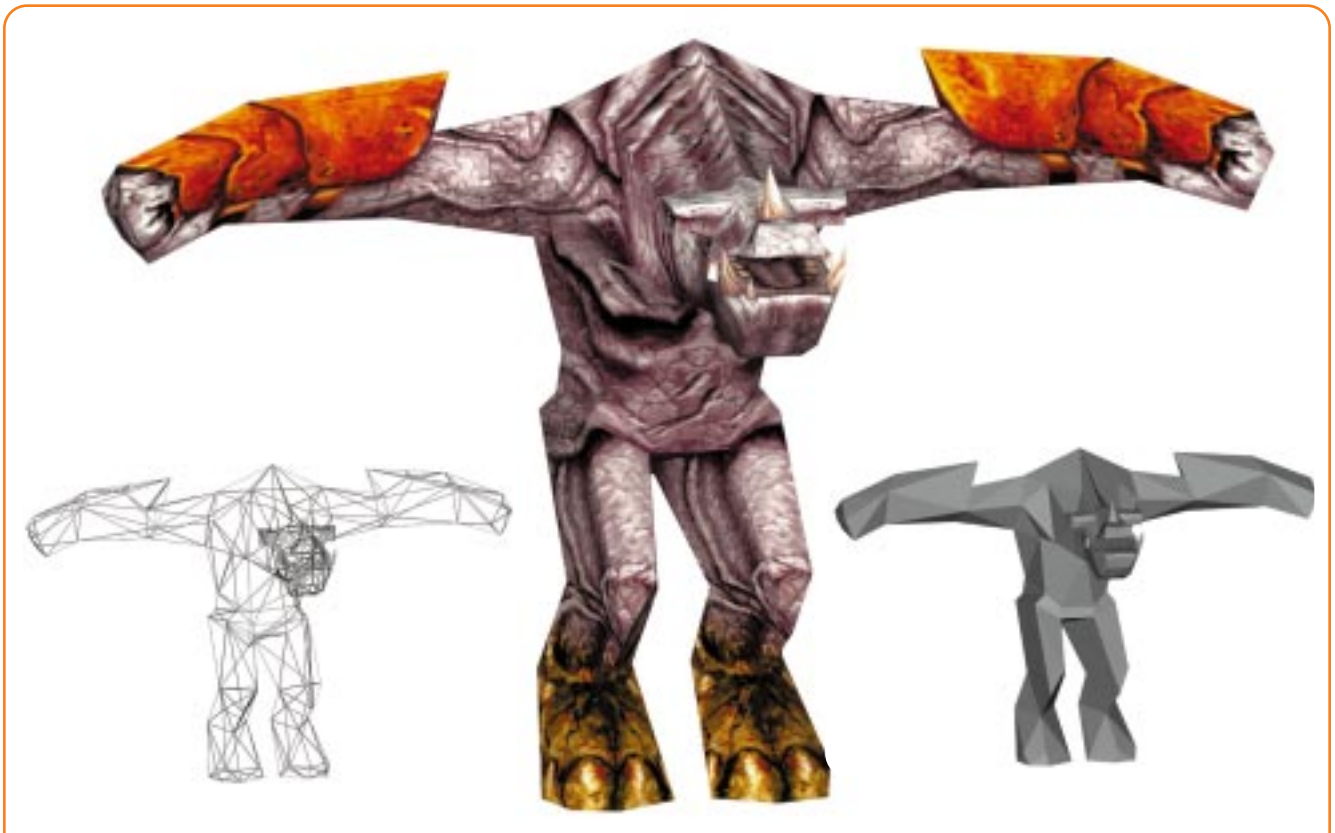
dreds of megabytes of data on our minimum system requirement of 32MB RAM. The two main characters, Rynn and Arokh, total more than 20MB of animations, plus 12MB of sounds (including in-game cutscenes). To pull this off, the system keeps the most recently used sounds or animations in the cache and can flush memory that it hasn't used in a long time. Further reduction of memory usage is achieved by sharing animations between characters with the same skeleton, even if they have completely different skins. The system only loads the data that it needs, as it needs it. This is important during development, as artists and designers are prone to leave unused textures, sounds, and models in a database. The result is good engine performance during development, which is also representative of the final product.

We tried to ensure that the engine and tools always displayed to the artists and designers something that was representative of the final game (WYSIWYG). The best example of this was our real-time 3D editing system. The engine was integrated into the editor, so any geometry, texture mapping, or lighting changes made by the designer would be immediately reflected in the 3D view. The importance of this aspect of the tools should be emphasized because it gave the designers the ability to tune levels and game play very quickly and with a minimum of guesswork.

Concept drawing for DRAKAN's mountain world.

4. COMPELLING DESIGN. A good design will not only sell a game — it can also help smooth the development process. The DRAKAN world has immense possibilities, so new ideas were born easily within its scope. This kept the team highly motivated, as there were always innovative things to do with the genre. The varied environments gave a wealth of new things to work on for the art and design team, and were an ideal canvas for programmer invention.

The design also kept Psygnosis very interested. DRAKAN became its top PC product, and it was comforting to us as developers to know that our publisher was behind the product. Psygnosis saw the marketing potential in a beautiful female character combined with a fearsome, fire-breath-



The war giant towered above Rynn and had multiple high-resolution texture maps.

ing dragon and the press latched on to the concept with excitement. They could market it to TOMB RAIDER fans, AD&D fanatics, and even 3D shooter addicts.

Even the most brilliant design would be difficult to implement lacking a proper design document. The 175-page DRAKAN design document contained outlines for the entire game, including all AI behaviors, weapons, and level flows. It served its initial purpose well, and was a blueprint for our lead designer's vision. The document was vital to the development team, especially when it came to scheduling, creating tasks, and communicating with the publisher. But as you will read in the following "What Went Wrong" section, feature creep overtook the project halfway through, and the document never kept up with the changes. A design document should always be maintained throughout development to preserve it as a useful resource for the team. Fortunately, the team could always rely on Alan to explain anything or to fill in any holes in the design document.

Colored conceptual sketch of a Wartok grunt.

Concept sketch of the Dark Union sailing ship.

5. INDOOR/OUTDOOR ENVIRONMENTS. One of the major technologies that set the Riot engine apart from the other landscape engines was its ability to render both indoor and outdoor environments using the same engine. The benefits to game play were huge because we could do arbitrary cave systems, arches, overhangs, and other structures that were perfect for a fantasy game. The "layer" system that the landscape was created with was ideal for massive outdoor environments and allowed the designers to create very organic-looking worlds. Rectilinear structures such as buildings and objects were created using arbitrary models imported from external 3D modeling programs. Although these models were not included in the visibility calculations, the layers were included and were nicely suited for use in the visibility culling of large environments. Because the layers were small height fields that made up the ceilings and floors of the surroundings, they took up very little space in memory. This meant that the levels could be vast, and it helped give the player a sense that there was a living world around them.

What Went Wrong

1. STAYING ON SCHEDULE. DRAKAN was originally slated for release in February 1999, but ended up being released six months later. Even with careful scheduling and task planning, we failed to meet the final deadlines. Part of the problem was that we didn't account for the time the team would spend creating versions of the game for E3 and for magazine and Internet demos. Each demo pulled nearly two weeks of time away from our normally scheduled tasks. The majority of the scheduling problems were due to feature creep and other improvements that were considered necessary during development.

In March 1998, the design team was faced with a mountain of work ahead in order to complete the 14 original levels as designed. After careful consideration, the designers decided to spend their efforts on enlarging and improving upon the ten best level designs. They also ended up cutting many features that did not show much game-play promise. The dart gun and the boomerang weapons were among those eliminated from the game. Even though these tasks had already been mostly completed (in terms of code) for several months, they had not yet been put into the game. By the end of the project, the designers did not have adequate time left to work with programmers to play-balance those features, and the art staff had not done any work on them either. So they got cut. The decision allowed us to focus on improving the weapons that worked well, such as the bows and arrows. We now know that it's critical that programming tasks get put into the game and tested almost immediately so that their effectiveness can be realized early on. This lack of coordination between designers, artists, and programmers often caused problems during development. Some of this was because our design document wasn't updated when weapons, levels, or AI were redesigned.

The initial AI programming followed the original design document, but didn't work well when put into the game. It wasn't until our designers worked with the AI programmers to figure out exactly what they wanted for our combat system that the AI really came together. This kind of collaboration should have occurred at the beginning of the AI programming process and the lack of it caused moderate delays.

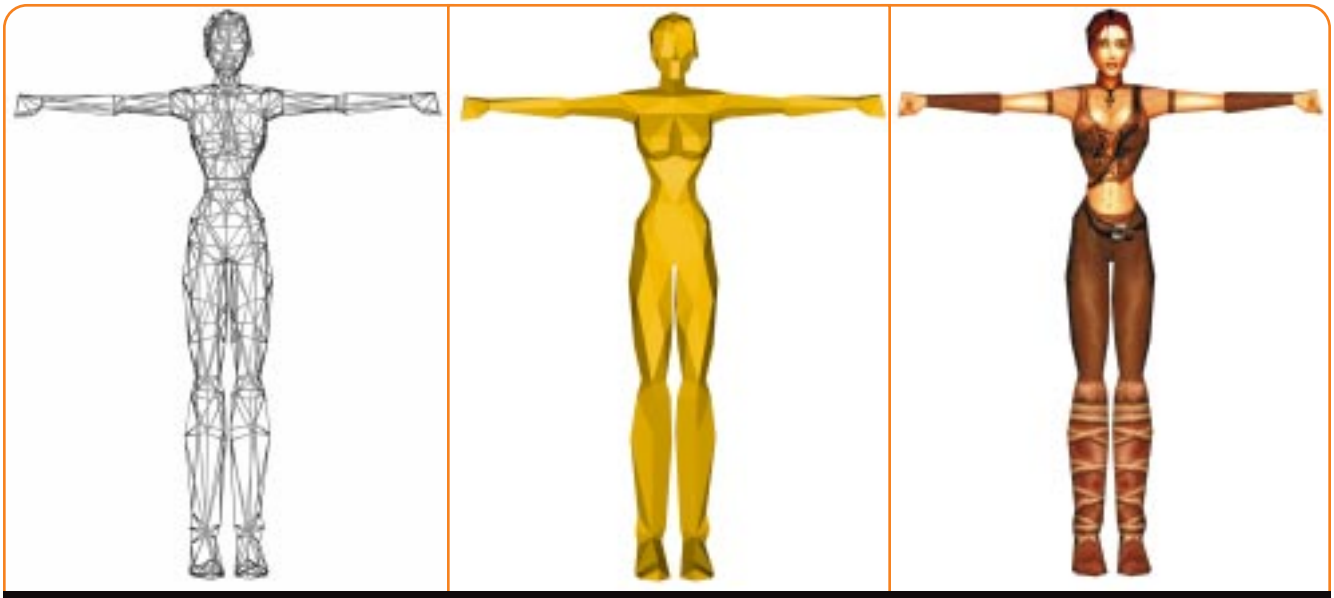
DRAKAN's art team often rebuilt geometry and model textures, sometimes up to three times before they were satisfactory. This may have been partially due to Surreal's high aesthetic standards, but a lack of consistent artistic vision is also to blame. DRAKAN had lots of character conceptual art, but no "art bible" to document all the models and environments for the game. This meant that if our art lead was not satisfied with work of another artist, he would often rebuild it himself. At various points during development his time was spread thin across many different tasks. In addition, the art team went through communication problems and power struggles that hampered the coordination of the team.

2. INADEQUATE TESTING. Although we tracked bugs internally before and during the alpha and beta releases, Psygnosis was responsible for the bulk of the testing after alpha. For a game as vast and ambitious as DRAKAN, the time that we allocated for testing was inadequate. Multiplayer and collision detection issues, in particular, were not given enough testing time. When the final shipping date approached, we reluctantly agreed to allow some noncritical bugs to slip through to the gold master in the interest of meeting the deadline. A patch was inevitable.

Other testing complications added to the problems. As Psygnosis was being reorganized by its parent company, Sony Computer Entertainment Europe (SCEE), half the testing department was let go and merged with SCEE's U.K. testing group. This caused minor hiccups in tester allocations to

Morphing walls were created by assigning special behaviors to the world geometry.

A multiplayer ground deathmatch level.



Rynn's highest polygon count was only 538. Single-skinned characters such as Rynn generally look much better with fewer polygons compared to the segmented characters traditionally used in most game animation systems.

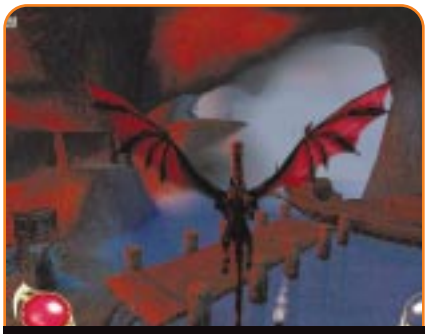
DRAKAN. Since the testing team was located in Europe, communication was difficult, and often messages were delayed by a day or two. Bug reports were sent to us via Microsoft Excel worksheets, which were converted from an Oracle database that sat isolated on their LAN in the U.K. Often the Excel worksheets would come to us corrupted or would have incomplete bug descriptions. Bug responses from Surreal's programmers had to be tracked carefully and entered back into the Oracle database by hand.

Dragon's-eye view of a fantastical island formation.

We kept our own internal database at Surreal using Outlook forms in special public folders on our Exchange Server. We ended up generating almost 1,000 internal design, art, and programming bugs during the entire project. This rivaled the number of bugs generated by the testing team during alpha and beta. The internal system worked very well, but it could have been more useful if the Psygnosis testers had access to the system as well. We tried getting on-site testers, and some of the U.K. team did come to Surreal for about a week. But it was too late in the project and for too short a time to be effective.

3. COLLISION DETECTION AND RESPONSE. One of the biggest chores for the testing team was to make sure that all of our hundreds of 3D models could not be penetrated by missiles, NPCs, or Rynn. Each model had to be properly bounded by the artist during the model's construction, a process which took about 20 percent of their modeling time to construct. Bounding was generated in our custom modeling tool and approximated the polygons of the model using a hierarchy (tree) of bounding spheres or oriented bounding boxes (OBBs). This made the collision detection system very fast and accurate, but it also meant that if an artist made a mistake in the bounding tree, collision detection might not work. To say this created a testing challenge would be an understatement.

Even though the engine was capable of rendering arbitrary meshes, the collision detection system was not designed to handle some of the detailed meshes



Lighting effects created varying moods. This bridge model was reused from the islands level.

that the artists produced. Some of our AI used the bounding information at the lowest level, while Rynn's collision response system used a polygon-accurate analysis, which didn't work perfectly for some complex models. Frame-rate variations across machines also caused differing results, making it hard for programmers to reproduce the bugs and correct the problems. Finally, our indoor/outdoor landscape system created some challenging collision-detection problems that we hadn't anticipated when it was originally designed.

4. MULTIPLAYER. Considered by some the Achilles' heel of DRAKAN, its multiplayer suffered from developmental neglect. For the game's multiplayer to have succeeded, the design, art, and programming teams would have had to spend at least twice as much time on it than they did. The two multiplayer designers did most of their level and weapon work during the alpha and beta periods. The same designers also created most of the artwork for the multiplayer effects and weapons. Any game-related bugs that came up were fixed by our single network programmer, who already had his hands full optimizing the underlying network engine. Most of these game-related problems arose because the same weapons were used in both single and multiplayer games, but the original programmers were not careful to make them "network aware."

Originally, we thought that DirectPlay was the easiest networking solution for us. But as the design got more complex, we found that DirectPlay just did not work well for us. DirectPlay was a debugging nightmare. The network programmer's machine crashed several times per day when debugging the networking code and we couldn't determine what was causing that to happen. It wasn't until we switched to Winsock that we discovered that the crashes were caused by DirectPlay. DirectPlay also caused a serious problem for us while we debugged the game under the first release of Windows 98. DirectPlay actually caused the system clock to slow down. This caused the game to run slower and sucked up tons of CPU cycles, forcing a reboot. When put to the test, DirectPlay also had issues with firewalls, which we were not able to resolve. Under certain circumstances, the way DirectPlay handled the message queues sometimes caused messages to pile up until the application hung. Perhaps Microsoft will be addressing these issues in future releases.

It was clear even before alpha that the networking code would need to be rewritten. In the final design, we only used the TCP/IP portion of DirectPlay, and we used a Winsock front end to handle communication with the master server. We proposed to Psygnosis that they give us more time to convert the system over to Winsock, to which they replied yes — but only as a downloadable patch, since the additional work would have delayed the game's release. The Winsock conversion was not finished until a month after DRAKAN's release, and greatly stabilized the multiplayer experience. This, com-

bined with the release of the level editor and mods, has created a resurgence in multiplayer support, but it will never be as good as it could have been.

5. Badly executed story. Although the overall story concept of DRAKAN was a great, the script and execution of the idea were lacking. We hired a movie scriptwriter to do the initial work on the script, but he was not familiar with the fantasy genre and did not have a firm grasp on Alan's vision for the design. From there, the script was edited and rewritten by several more people: members of Surreal, members of Psygnosis, even one of the voice actors. Under pressure to finish the script, it was completed with cheesy one-liners and other badly written dialog. Once it had been recorded by the voice actors, it was very difficult to re-record lines that were badly written or acted. Some were re-recorded, but that was a luxury we could only afford for the completely failed lines. The voice acting was difficult to get right, because just as some of the writers had almost no vision of the game, the voice actors likewise had little understanding of the characters they portrayed.

The island level showing the organic qualities of the terrain and textures.

Another problem with the execution of the story was that most of the construction of the cutscenes was left until the last minute, since all the levels had to be "geometry complete" before cutscenes could be created. This meant that the scenes at the end of the game were hastily done, and some even had to be cut from the game.

Onward to the Next Projects

DRAKAN's development was a bumpy ride, but it went suitably well considering it was the first game developed by an inexperienced team. Even with some of the schedule slips that occurred, the great design, art, and programming kept the project going strong. DRAKAN has been a great learning experience for the team, and the careful evaluation of our past mistakes has helped us in the development of our current projects. DRAKAN was recently named PC Game of the Year by several popular magazines, and it has sold very well. If DRAKAN showcases what this team is capable of in our first project, it will be very exciting to see what we are capable of in the future. ■

by Mike Sellers

Pay and Play and Pay: The Future of Online Gaming

By now the Internet has fully invaded most aspects of our lives. So what about online games? If you've been around the industry at all, you know that games have actually been online for at least a couple of decades. The problem is that what worked then doesn't work now: the Internet of 1984 bears zero resemblance to the Internet of 2000. And yet, too many people in the industry are still looking backward, assuming that little has changed, or what worked in single-player games will work with multiplayer games online. So the question for our industry is not "Hey, can we play games online?" but "Can we make lots of money by getting lots of people to play our games online?" The answer to the first question is a definite "Yes!" The answer to the second one is more problematic. It seems like an easy concept: People are flooding onto the Internet. People like games. People like playing games with other people. So it should be a slam-dunk to get people to play games with other people on the Internet and make scads of money in the process, right? Wrong.

The problem with making money from online games is that you need some way of actually getting someone to pay you for providing your games, preferably over and over again. Most small developers know how difficult it is to get a game onto the retail shelves, and even once you've made it there, how perilous it can be trying to sell it in a fiercely competitive market. The Internet, with its seemingly infinite space for games — not to mention its lack of publishers, distributors, or boxed goods — seems like it should be a paradise for developers.

illustration by Jackie Urbanovic

But the online business models can be harrowing. You can either try to get people to pay you to play your games, or you can try to get someone else to pay you for attracting people to play your games. This latter model is the one that everyone was looking at as a panacea just a year or so ago: Sell ads that people see while they play your games! What could be simpler? Unfortunately, this is not a stable part of the industry.

The presence of advertisements online has soared over the past couple of years, but inventory (places where ads could be shown) has exploded even faster. As a result, the price that you can charge for providing ad space has gone through the floor. CPMs (cost per thousand viewings) have decreased from $30–$50 to just $2–$10, and click-through rates have stabilized at around 1–2 percent of ads shown. Moreover, while click-through rates for people playing parlor games tend to be about the same as for those doing other online activities, the more compelling the game, the less likely its players are to take a time-out and go hit a hot ad.

This all means that it is incredibly risky to try to base a business strictly on ad revenues. And as advertising on the Internet matures, this problem is going to get worse, especially for the small developer. There are more high-profile places an advertiser can attract customers, and free games are increasingly a backwater in online traffic.

Which brings us back to getting people to pay you directly to play your games. The now-standard model for pay-for-play online is to charge customers $10 per month to play all they want, often in addition to paying

Continued on page 71.

Mike Sellers was the designer of MERIDIAN 59, the first commercial 3D massively-multiplayer online game, launched in 1995 (or about a hundred years ago, Internet-time). He has co-founded two online game companies, and now works for Maxis doing cool online stuff. He can be reached at archetypist@hotmail.com.

Continued from page 72.
for a game at retail. Other models have been tried, but no other serious challengers are yet on the horizon. This model can reap great financial rewards for the game publisher, though it carries with it increased and sometimes unforeseen responsibilities, too. In online games, the product is just the tip of the iceberg: what you're really selling is a service. If, as a developer or publisher, you're not prepared to run a service and support a community, you should steer away from providing online games.

All this sounds pretty gloomy. Make no mistake, though, online gaming is gaining steam and is going to be huge. But ad-supported games are a rickety, risky business at best. And in pay-for-play online games, the big winners are going to be the companies with exclusive and established brands, both for PCs and eventually consoles. (When console games finally venture online, there will be a whole new host of design and technical issues to deal with, but given the number of console owners, the rewards could eclipse anything the game industry has seen yet.)

So what's a small developer to do? First, lose any remaining starry-eyed naïveté. The Internet is a gold rush, but for every 28-year-old Internet millionaire, there are ten 40-year-olds with third mortgages. If you're set on moving into online games, understand that you're by no means alone (there are more than 70 massively-multiplayer game projects underway right now by one estimate), but neither are you too late. This area is the new frontier for the gaming industry, but it isn't for the faint of heart, or those with just another implementation of Hearts. ■