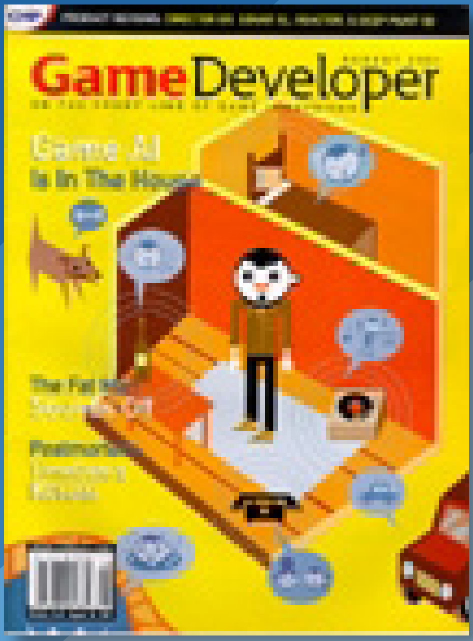
GAME DEVELOPER MAGAZINE

AUGUST 2001



UBM Tech

Pivot

We've reached a turning point in the game development industry. We're not the hacker culture we once were. You're not making games in your basement. Suddenly you're scouring research papers to get a leg up on your competitors, you're using software engineering tools to describe design patterns to your peers, and you're hiring Ph.D.s fresh out of school to work on your physics simulation. Orchestras and top-40 bands are on your soundtrack. You have someone dedicated to landing license agreements for advertisements inside your game. And you're directing sports stars in motion capture shoots and Hollywood starlets for your voice-overs.

What Happened?

Games have become big business. Over one billion dollars will be spent marketing game consoles this year. Microsoft has joined the fray. RealNetworks is distributing games in addition to audio and video. WildTangent recently landed $34 million in third-round funding, largely based on a revenue model of making money through games as advertisements. Nokia is courting game developers for cell phone titles. Ericsson was building a portable game device with cellular features. Even Sun is building a Java Game API. Everyone wants games.

In this new, new world of game development, the old paradigms aren't working. You can't just get together with a few of your closest friends and create a top-tier title. There's big money to be made out there, but making it takes a lot of risk, and that necessitates a lot of planning.

For the new game studio this means exclusive deals, licensing arrangements, royalty advances, and extensive reviews with the publisher. For you day-to-day this means more pre-production work, design documents, software engineering, use of design patterns, reusable components, and code reviews. If you've been in the industry for a while, this new landscape may not be very appealing. With big money it's harder to take a big risk. The creativity of the industry slowly bleeds away.

But with these new challenges comes a huge crop of young and excited game developers, raring to put your new title on the shelves. Game development is being taught around the globe, at both reputable universities and dedicated schools.

Suddenly these new graduates, and their research, are very applicable to game development. Current graphics research is particularly valuable, as even the venerable Siggraph conference has slowly turned toward pushing real-time graphics as the new frontier.

Looking for New Models

But graphics aren't the only area where you can find value outside our industry. Hollywood has a lot to offer us. Motion capture studios, camera techniques, lighting and set design, storytelling, and music composition are all areas that Hollywood has been studying for years, and they are directly applicable to your games. If you're not tapping TV and film resources, you're falling behind.

Some say that copying Hollywood's way of doing things will result in our industry losing its uniqueness. They say that we shouldn't copy Hollywood, lest we end up like Hollywood, full of big money players who take no risks, and with a seedy underbelly. But consider that everyone in the world near a TV or theater is already familiar with the "language" of how Hollywood tells a story. These people understand camera angles and the nuances of a film score, and they are your new audience. Speak to them. It is perhaps unavoidable that the game development industry will generate its fair share of libel suits, velvet ropes, licensing battles, contract fights, scandals, and *National Enquirers*. But consider that these are always attracted to a successful, popular, high-money industry. With any luck, in another 10 years, you'll be looking back at today and saying, "I remember when …"

Mark

GameDeveloper
600 Harrison Street, San Francisco, CA 94107
t: 415.947.6000 f: 415.947.6090 w: www.gdmag.com

Publisher
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL
Editor-In-Chief
Mark DeLoura *mdeloura@cmp.com*
Senior Editor
Jennifer Olsen *jolsen@cmp.com*
Managing Editor
Laura Huber *lhuber@cmp.com*
Production Editor
R.D.T. Byrd *tbyrd@cmp.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Daniel Huebner *dan@gamasutra.com*
Jeff Lander *jeffl@darwin3d.com*
Tito Pagán *tpagan@w-link.net*
Advisory Board
Hal Barwood LucasArts
Ellen Guon Beeman Beemania
Andy Gavin Naughty Dog
Joby Otero Luxoflux
Dave Pottinger Ensemble Studios
George Sanger Big Fat Inc.
Harvey Smith Ion Storm
Paul Steed WildTangent

ADVERTISING SALES
Director of Sales & Marketing
Greg Kerwin e: *gkerwin@cmp.com* t: 415.947.6218
National Sales Manager
Jennifer Orvik e: *jorvik@cmp.com* t: 415.947.6217
Senior Account Manager, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 415.947.6224
Account Manager, Recruitment
Morgan Browning e: *mbrowning@cmp.com* t: 415.947.6225
Account Manager, Northern California
Susan Kirby e: *skirby@cmp.com* t: 415.947.6226
Account Manager, Western Region, Silicon Valley & Asia
Craig Perreault e: *cperreault@cmp.com* t: 415.947.6223
Sales Associate
Aaron Murawski e: *amurawski@cmp.com* t: 415.947.6227

ADVERTISING PRODUCTION
Vice President, Manufacturing Bill Amstutz
Advertising Production Coordinator Kevin Chanel
Reprints Stella Valdez t: 916.983.6971

GAMA NETWORK MARKETING
Senior MarCom Manager Jennifer McLean
Marketing Coordinator Scott Lyon
Audience Development Coordinator Jessica Shultz

BPA INTERNATIONAL
Game Developer magazine is BPA approved

CIRCULATION
Group Circulation Director Catherine Flynn
Director of Audience Development Henry Fung
Circulation Manager Ron Escobar
Circulation Assistant Ian Hay
Newsstand Analyst Pam Santoro

SUBSCRIPTION SERVICES
For information, order questions, and address changes
t: 800.250.2429 or 847.647.5928 f: 847.647.5972
e: *gamedeveloper@halldata.com*

INTERNATIONAL LICENSING INFORMATION
Mario Salinas
t: 650.513.4234 f: 650.513.4482
e: *msalinas@cmp.com*

CMP MEDIA MANAGEMENT
President & CEO Gary Marshall
Executive Vice President & CFO John Day
President, Business Technology Group Adam K. Marder
President, Specialized Technologies Group Regina Starr Ridley
President, Technology Solutions Group Robert Faletra
President, Electronics Group Steve Weitzner
Senior Vice President, Human Resources & Communications Leah Landro
Senior Vice President, Global Sales & Marketing Bill Howard
Senior Vice President, Business Development Vittoria Borazio
Vice President & General Counsel Sandra Grayson
Vice President, Creative Technologies Philip Chapnick

GamaNetwork
A DIVISION OF CMP MEDIA LLC
WWW.GAMANETWORK.COM

Return

SAYS YOU

THE FORUM FOR YOUR POINT OF VIEW. GIVE US YOUR FEEDBACK...

Reader Questions Artificial Lag

The lag compensation technique described in Yahn Bernier's article, "Leveling the Playing Field" (June 2001), has a problematic drawback that was not mentioned in the article — artificial lag. For example, suppose an enemy is running away and will hide behind a corner in the next moment. If now, by a cheat hack, the client is prevented from getting further updates from the server, the cheating player has somehow "stopped time." He has all the time in the world to run behind the enemy, target the enemy, and press the fire button. When the server gets these latest movement and fire commands, it calculates back to the state at the stopped time where the enemy was not behind the corner yet and calculates the hit by the gun. For this cheat, in principle you only need a simple switch to suspend incoming networking packages. Maybe in HALF-LIFE, for example, they were able to avoid this particular cheat by not allowing a sudden change of the ping time.

On the other hand, this is a more fundamental problem with this kind of lag compensation technique. There seems to be in principle no way to protect against the following cheat: two players in a team sit next to each other, but player A plays with an additional artificial lag of one second relative to player B. Then the screen of player B somehow shows the future of player A one second ahead. If on screen B an enemy performs some particular action, player A knows that with a one-second delay the enemy will perform the same action on his screen, and he can perform all preparations to fight the enemy in the best way, since he knows the "future."

Folker Schamel
via e-mail

YAHN BERNIER RESPONDS: *The "artificial lag" technique is something we were aware of during the design and implementation of lag compensation. There were three things we did to reduce the effectiveness of the lag technique. First, we placed an absolute limit on the amount of time for which we would allow lag compensation. The default compensation time for HALF-LIFE and its mods is 500 milliseconds. Even if the user freezes their net connection with artificially created lag, it would be hard to move entirely behind the enemy in that small amount of time. Second, we allowed server operators to dampen the latency measurement for players on the server. By sampling latency over several packets, any large changes in latency caused by creating lag are smoothed out over time. Finally, since the HALF-LIFE engine contained a built-in way to simulate latency, we ended up easing in and out of simulated latency in our system and, later, making the latency simulation command available only to clients playing on servers with cheats enabled.*

Article Targets Wrong Audience

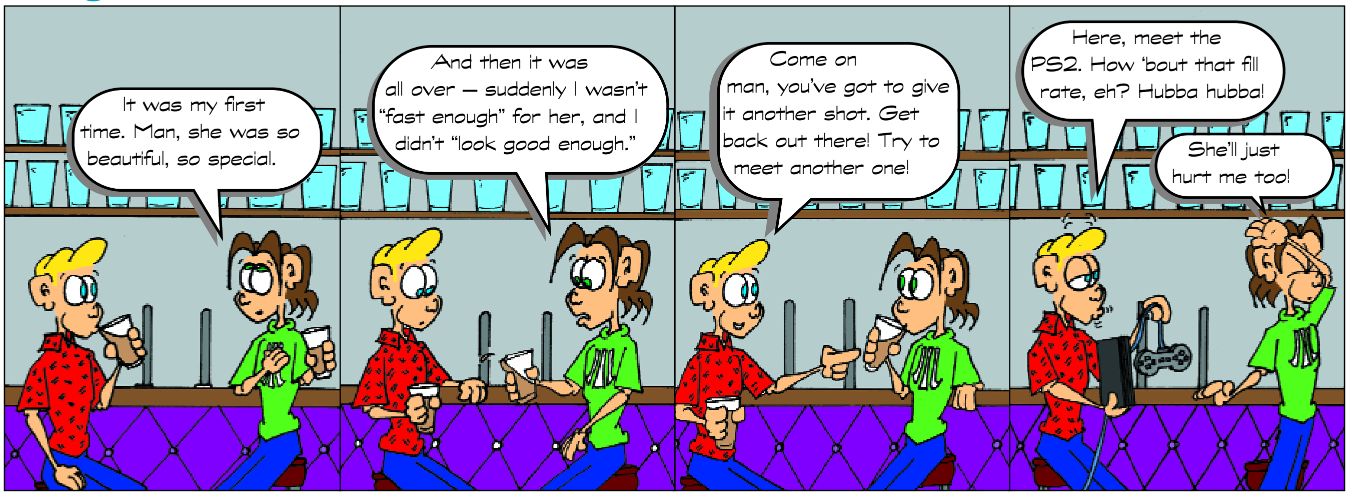
The comments about software patents in *Game Developer* ("Software Patents Should Be Abolished," Soapbox, May 2001) are true; however, if you are trying to convince software engineers and designers, you are targeting the wrong folks. Software patents are not about protecting innovation, they are about lawsuits and money. I don't mean that most companies pursue patents to file lawsuits — they pursue them to protect themselves from other companies. It's like an arms race. We have to arm ourselves because the other guys are arming themselves.

Roger Collins
via e-mail

CHRIS AND CASEY RESPOND: *Some companies and individuals use patents offensively, not defensively, and we mean "offensively" in every sense of the word! And even for those that only use patents defensively, if even a small part of the effort of attaining patents was put toward reforming or dismantling the patent system, the world would be a better place. Burying one's head in the sand while muttering, "Everybody else is doing it," is not a recipe for improving the situation.*

Send e-mail to editors@gdmag.com, or write to *Game Developer*, 600 Harrison St., San Francisco, CA 94107

Kludge by Tiger Byrd and Daniel Huebner


It was my first time. Man, she was so beautiful, so special.
And then it was all over — suddenly I wasn't "fast enough" for her, and I didn't "look good enough."
Come on man, you've got to give it another shot. Get back out there! Try to meet another one!
Here, meet the PS2. How 'bout that fill rate, eh? Hubba hubba!
She'll just hurt me too!

Return

INDUSTRY WATCH

daniel huebner | THE BUZZ ABOUT THE GAME BIZ

Microsoft acquires Ensemble Studios. Microsoft has acquired Dallas-based Ensemble Studios. Ensemble and Microsoft have a long-standing publishing relationship, and collaborated on the popular AGE OF EMPIRES series. Ensemble will continue to operate from its Dallas headquarters; the terms of the deal were not disclosed. Ensemble is the latest in a string of Microsoft game-studio acquisitions, joining the likes of FASA Interactive of MECHWARRIOR fame; Access Software, known for the LINKS golf simulation games; and Bungie Software, creators of the MYTH series, ONI, and the forthcoming HALO.

AGE OF EMPIRES, Ensemble Studios' popular real-time strategy game. Ensemble was recently acquired by Microsoft.

Sony and AOL form PS2 partnership. Thanks to a new partnership between Sony and AOL, Playstation 2 owners will soon be able to access the web and e-mail through AOL's online service. AOL features, including e-mail and real-time chat, will become part of Sony's PS2 SDKs and made available for developers to incorporate into their games' designs. Sony hopes to have this functionality built into the SDK by this winter, meaning it will be at least six to 12 months after that before the first AOL-enhanced PS2 games appear at retail. Also in the works to expand PS2 online possibilities are a Netscape browser and a version of the Real Networks media player.

Mixed financials. With the console transition nearing its final stages, some struggling companies are seeing the light at the end of the tunnel. For its part, Activision managed to beat analyst predictions in its fourth quarter. Revenues for the period were $127 million, up 22 percent from $104 million in the same period last year. Earnings for the period added up to $875,000, compared with a net loss of $53 million in the fourth quarter one year ago. The promising results have led Activision to raise its full-year 2002 per-share earnings estimates.

Electronic Arts also managed to beat estimates in their fourth-quarter results, posting a smaller-than-expected loss. Consolidated net revenue for the fourth quarter was $307.3 million, up from $294.3 million in the same period last year. The company's loss for the fourth quarter was $17.9 million, compared with net income of $3.4 million for the fourth quarter last year. The company incurred a one-time pretax charge of $2.7 million in the quarter related to the acquisition of Pogo.com.

The picture is less rosy at Eidos. The company is reporting revenue of $32.1 million and a quarterly loss of $38.3 million. Looking to secure new capital, Eidos is planning to offer existing shareholders a bargain on new shares. The company plans to raise more than $70 million by offering its shareholders the option to buy into the new issue at a price substantially below current value. Eidos holds that the issue is intended to raise funds to finance new development projects, but following the company's recent financial woes a share issue could also help erase debt as Eidos looks to return to profitability in the coming year.

Sega is also looking for ways to tap new capital. Extraordinary costs related to the Dreamcast shutdown are forcing Sega to issue bonds. The company will issue more than $400 million worth of zero coupon bonds in order to offset Dreamcast debt and costs related to development for rival consoles.

Xbox and Gamecube set dates, announce pricing. Both Microsoft and Nintendo took advantage of the press gathered at E3 to unveil launch dates and pricing for their next-generation game consoles. Microsoft set November 8 as the launch date for Xbox and announced that it will be priced at $299. Microsoft said it expects to sell as many as 1 million to 1.5 million units during the holiday season. Just hours later, Nintendo fired its first volley in what is shaping up to be a true console war between Nintendo and Microsoft. Nintendo set the U.S. Gamecube launch for November 5, three days earlier than Xbox. Nintendo upped the ante even more, announcing several days later that Gamecube will come to market at $199, $100 less than the $299 Xbox price tag.

Havas Interactive changes name to Vivendi. Havas Interactive, the parent company of Sierra, Blizzard Entertainment, Universal Interactive Studios, and Knowledge Adventure, has changed its name to Vivendi Universal Interactive Publishing. The name change standardizes the Vivendi name across the company. Vivendi Universal Interactive Publishing is itself a subsidiary of Vivendi Universal Publishing. The newly named company is made up of two operating divisions, Vivendi Universal Interactive Publishing North America and Vivendi Universal Interactive Publishing International, each controlling the sales, distribution, and operations within its region.

UPCOMING EVENTS CALENDAR

GDC EUROPE
ExCeL
London, U.K.
August 31–September 1, 2001
Cost: £411.25 before July 31; £528.75 after July 31
www.gdc-europe.com

ECTS
ExCeL
London, U.K.
September 2–4, 2001
Cost: Advance registration free via web site; £25 on-site registration
www.ects.com

Return

Macromedia Director 8.5 Shockwave Studio

by brian robbins

With Director 8.5, Macromedia promises developers the ability to deliver dazzling 3D entertainment on the web. Not only do they deliver on this promise, but they have exceeded many people's expectations. Using Director 8.5, developers can create 3D web-based games that are small, run reasonably fast, and operate cross-platform for Windows and Mac.

Obviously, the biggest change from Director 8.0 to 8.5 is the addition of the 3D rendering engine. However, in addition to the 3D engine, Director 8.5 and the new Shockwave Multi-User Server 3.0 support UDP messaging. This is the first step toward being able to offer truly real-time multiplayer gaming through Shockwave. Director 8.5 now also supports embedded Flash 5 files, and RealVideo through Shockwave.

Director 8.5 offers very easy access for developers to create 3D content. Using Director's built-in programming language (called Lingo), developers can create four basic types of primitives: sphere, box, cylinder, and plane. However, possibly the most useful feature is the ability to create an arbitrary mesh via Lingo. This allows developers to create any object by passing in the vertex data for the faces. The possibilities for this feature are practically limitless. One developer has even created a tool that reads in and creates the geometry for a QUAKE level using this technique. Another interesting feature which can be accessed without requiring modeling software is particle systems. While this is not a necessary feature for any 3D engine, it is a very welcome one. The particle systems run reasonably fast and provide an easy way for developers to create effects such as smoke, fire, sparks, and so on.

The true power of Director 8.5 is seen when models are created in modeling packages and imported into Director. It supports the new Shockwave3D, or .W3D, file format. Several companies have committed to creating exporters for this format. At the time of this writing, Discreet has an exporter available for 3DS Max 3.1 and 4.0, Alias|Wavefront has an exporter for Maya 3, and Caligari has included their Shockwave 3D exporter into TrueSpace 5.1. According to Macromedia, exporters for Lightwave and Softimage are in beta testing or ready for release soon.

An image created in Macromedia's Director 8.5 Shockwave Studio.

Aside from not requiring artists to learn a new program to create models for Director 8.5, another benefit of using .W3D files is access to modifiers. Seven modifiers are provided which greatly enhance the abilities of Director 8.5:

Level of detail (LOD). This can reduce the number of polygons interactively based on how far the object is from the camera, or to achieve an author-specified target frame rate. Setting this up requires a bit of tweaking to get right, but the benefits can be well worth the effort, especially on older systems.

Subdivision of surfaces. This does the opposite of level of detail, adding polygons to objects and theoretically making the object look better. This can do a lot for smoothing out round objects, but in most cases this feature will probably not provide a large enough benefit to be used extensively.

Keyframe and bones player. These provide methods for playing keyframe and bones animations created in a 3D package and are the easiest way to incorporate animations. Both modifiers support animation blending, which enables smooth transitions between animations. This is a very powerful technique that allows character animators to set up different motion cycles, such as a walk and a run cycle, without having to worry about the transitions between them. Several very impressive demos have been created with characters modeled and animated in 3DS Max and Character Studio. The big advantage here is that new bones animations can be exported without the accompanying geometry data. The file size for these animations is very small, so many animations can be included without a major file-size hit.

Mesh deform. This modifier provides access to every vertex and face in a model, as well as the normals and texture coordinates. As the name implies, it also allows the deformation of a mesh. This modifier can be used to create impressive rippling water effects.

Toon and inker. An interesting new technology that has yet to find a good use is non-photorealistic rendering (NPR). The toon and inker modifiers will render models with a cartoonlike look. While there is a noticeable performance hit for doing this, the effects are quite astounding.

While Director 8.5 is designed to provide a general 3D rendering engine for many diverse applications, the most compelling application is for game development. With this in mind, we set out to create a typical first-person shooter (FPS). Sure, there are much more innovative things to create than an FPS, but we figured if Director 8.5 can handle a decent FPS, then it should be able to handle just about anything for games.

In order to utilize the streaming functionality, as well as to test importing mul-

★★★★★ excellent
★★★★ very good
★★★ average
★★ disappointing
★ don't bother

tiple models into one world, we created our levels out of "building block" models pieced together to create hallways and rooms. The advantage to this method is that 3D modelers can create the pieces and rely on the programmers to create the levels, as opposed to forcing the 3D modelers to create the levels. However, the biggest drawback is that cloning the separate models into the 3D world and then subsequently duplicating them to actually build the level takes more time than simply opening a complete .W3D level. This necessitates the use of the same loading screen that you see in almost every 3D CD-ROM game produced.

One drawback to Director 8.5's general-purpose 3D engine is that it does not know anything about the world it is displaying and thus cannot make any display optimizations. This means that the developer should do a lot of performance optimizing, such as removing models from the world that aren't visible. Fortunately, the methods for adding and removing models are extremely fast, and using them effectively will drastically increase performance. This drawback is a necessary fact of life. While the engine in our case is being used to create games, it needs to be able to handle all types of applications, from product demos to business and marketing presentations.

We also put Director's streaming ability to the test with this game. The initial file download is just over 400KB. This contains everything needed for the first level of the game. The remaining 400KB of data is streamed in while the user is playing. Not only is our entire five-level FPS less than 900KB total, but half of it streams in while playing. While this takes a few minutes on a modem, it is still very playable. That's a big difference when compared to the 50-plus-MB game demos for most CD-ROM games.

The overall performance of the Director 8.5 rendering engine is commendable. With our first-person shooter, we typically have between 2,000 and 3,000 polygons visible at a time, and we are able to achieve approximately 30 FPS with a 450MHz Intel Pentium II processor and a 3dfx Voodoo 3 card.

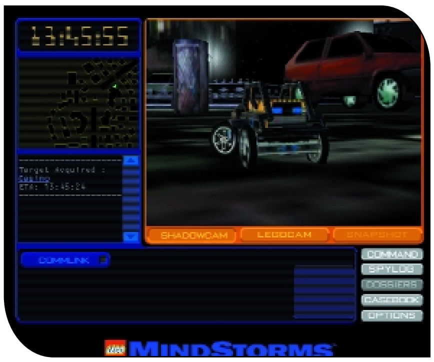
My favorite feature, and the one that I believe will truly set Director 8.5 apart from the competition, is the inclusion of Havok's rigid body physics. While currently not providing the full physics constraints provided in Havok Hardcore, Havok has made available rigid body collision detection and resolution, as well as dashpots and springs. As with everything developed in Director 8.5, this is available royalty-free for all projects.

The Reactor review that appears in this issue will give a much better idea of exactly what Havok's physics engine can do in Max, but for Director developers, this functionality far exceeds anything else available. The Havok data can be created entirely with Lingo, or the 3DS Max plug-in can be used and the results imported into the world. As a test of the Havok engine, I created a Newton's Cradle–type application and a simple ball rolling on a plane. Both of these demos were working in less than four hours and behaved almost exactly as you would expect them to.

For straight web-based game development, the main competitors of Director are Flash and Java. Flash 5 contains a moderately robust programming language, and developers can use it to create fairly complex games. However, Flash has a long way to go before it can approach the speed and complexity provided in Shockwave. Java has a development environment that can be obtained for less money than Director; however, Director more effectively supports multimedia natively, and provides a robust development environment.

For 3D web-based games, the main competition for Director 8.5 is WildTangent. Director has several advantages over WildTangent's technology. Director and Shockwave are cross-platform, and are, according to Macromedia, already installed in over 60 percent of web browsers and require no licensing fees for use. The cost for publishing a single commercial title with WildTangent technology is more than the cost to purchase Director 8.5. Director 8.5 adds the functionality of Havok's physics engine and an environment that is already familiar to many developers.

For developers already using Director 8.0 and not looking to create 3D content, there is no compelling reason to upgrade. Numerous minor improvements and bug fixes have been made, but none of them is so critical as to demand an upgrade. However, any developer looking at creating 3D content on the web, whether they have used Director before or not, should seriously consider Director 8.5. It offers an astounding number of features that work very well at a very reasonable price.



Lego's ROBOHUNTER, created in Director 8.5.

DIRECTOR 8.5 ★★★★

STATS

MACROMEDIA
San Francisco, Calif.
(415) 252-2000
www.macromedia.com

PRICE
$1,199 per platform. $199 Director 8 to Director 8.5 upgrade. $399 Director 5–7 to Director 8.5 upgrade. 90 days of technical support beginning with first call.

SYSTEM REQUIREMENTS
Windows 95/98/ME or NT 4/2000, or Macintosh OS 8.1 or later. 64MB of RAM, 100MB of disk space, 800×600 resolution display or greater, 3D accelerator recommended.

PROS

1. Solid 3D engine by proven company.
2. Havok provides world-class simulation.
3. Support from leading 3D vendors reduces learning curve.

CONS

1. Exporters not yet available for all 3D development products.
2. 3D on the web not yet successful.
3. Significant learning curve for developers unskilled in 3D.

★★★★★ excellent
★★★★ very good
★★★ average
★★ disappointing
★ don't bother

CAKEWALK'S SONAR XL

by andrew boyd

Sonar XL is Cakewalk's new flagship digital audio sequencer. It appears in place of Pro Audio 10 — apparently Twelve Tone Systems, makers of the Cakewalk line, felt that the brand was due for a freshening. Having reviewed Pro Audio 8 for *Game Developer* (March 1999), I was quite interested to see if more than just the name had changed in the last couple of years. After working with Sonar, it certainly does feel like they've taken the software in a new direction — and a good one at that.

I installed Sonar on a 700MHz Pentium III running Windows 2000 with a Sound Blaster Live! doing MIDI and audio duty. Installation was quick and easy, and featured automatic migration from Cakewalk Pro Audio 8, and automatic detection and configuration of the sound hardware. Performance was consistently excellent, even with lots of tracks and effects running. In true Cakewalk tradition there is a thorough and helpful printed manual provided and deep, easy-to-use online help. But neither should be necessary, as operation is impressively intuitive.

In use, Sonar looks more like Sonic Foundry's Vegas than like previous Cakewalk products. This is not necessarily a bad thing, as the Pro Audio line looked a little rough. Sonar has a slick, modern feel, and workflow is quite smooth. Most of the work in Sonar takes place in the Tracks window, which should be comfortable if you've ever used any multi-track audio software. MIDI and audio are displayed together in the right-hand pane of this window along with easily editable automation and controller data. Like Vegas, track information and mixing tools are in the left pane. All the volume, pan, mute, solo, effect inserts and sends, and metering are available for display right alongside the track. Tabs across the bottom of the left pane of this window allow you to quickly reduce the amount of information to common configurations — Mix, FX, and I/O. A traditional mixer window can be opened if you prefer to work that way.

It is evident that Sonar maintains Cakewalk's reputation for deep and robust MIDI functionality. As with other high-end sequencers these days, you would have to work hard to find a MIDI editing chore that is not easily accomplished in Sonar. Its audio implementation is similarly thorough — Sonar may be "version one," but it is essentially the tenth generation of Cakewalk's audio sequencer, with all the maturity that implies. Sonar also features a few extras that set it apart, such as native Acid-style loop tools and a new architecture for plug-in software synthesizers.

Sonic Foundry's Acid has become an industry standard tool; so much so, in fact, that Sonar's ability to import "Acidized" loops is a selling point of the software. Cakewalk's web site even brags that Sonar's loop tools make it a replacement for Acid. Not quite. For one thing, the independent time and pitch manipulation processes simply don't sound as good as they do in Acid. Many loops I tried worked fine, but in others I could hear artifacts not evident when using Acid. And, because Sonar is a big product designed to do a lot of things, it doesn't have the fun simplicity that Acid does. But don't get me wrong, the inclusion of this functionality gives Sonar a new level of flexibility. It's a wonderful addition to the package, and it does work well much of the time.

Sonar also debuts Cakewalk's DirectX Instruments (DXi) plug-in architecture. The world probably didn't need another plug-in format, but DXi does seem to work well and already has impressive support announced. Sonar XL installs several DXi instruments and a tube-amp simulator called ReValver SE DXi. ReValver is not an "instrument" as such, but what Cakewalk calls a DXi "transformer." This is basically a plug-in much like the DirectX processors also supplied with Sonar, but it offers automation through MIDI like an instrument. This is a great idea that I expect will catch on. None of the bundled effects or instruments blew me away, but all were quite serviceable, and they make up a very satisfying package.

If you're in the market for a Windows-based digital audio sequencer, Sonar should be high on your list of products to check out. It's got all the expected features, a great new interface, and some really exciting extras.

★★★★ | SONAR XL
Cakewalk | www.cakewalk.com

DISCREET'S REACTOR

by david wu

Take a game developer with an in-house physics engine, ask them to build their editor as a plug-in for Max, and you will end up with an offering in the spirit of Reactor. Reactor appears to have been designed from the ground up as an interface to the various features and concepts of the Havok physics engine. Its performance is outstanding — comparable to that of a mature game physics engine.

While not as well integrated as a system such as Maya, the 3DS Max/Reactor combination compares favorably to a good game editor. If you are comfortable with Max and you have both a basic understanding of physical modeling and some time on your hands, you should be able to put together your own dynamics-enabled scene. After I was comfortable with Reactor I had a lot of fun experimenting with it; in fact, I enjoyed playing with Reactor more than most games out there. I did find that if you are not careful, you can break the simulation — as my art director puts it, "Physics are not to be toyed with."

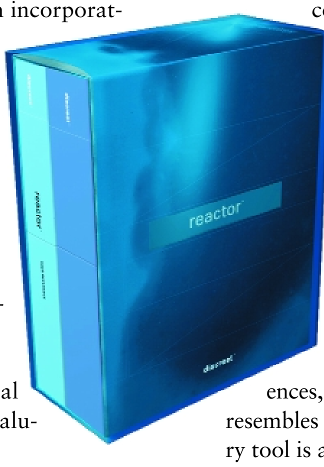
Reactor provides a suite of discrete co-existing technologies that collaborate to provide a powerful framework for integrating secondary dynamics into a scene. While it is possible to model the dynamics of characters and moderately complex vehicles, the level of control that you need in a game or cutscene is not something that is supported out of the box. Unless you are willing to write a highly complex controller in a limited environment, inanimate mechanisms such as boxes, barrels, dice, ropes, and ponytails form the scope of your dynamics-enhanced arsenal.

The strengths of Reactor include a stable rigid-body dynamics model, consistent collision detection and resolution, and real-time performance. While their rigid-body dynamics does not look quite right, it is highly robust and there exist many ways in which to customize and refine physical behavior. While less efficient than what you might find in a collision detection system designed for a specific game with specific constraints, Reactor's collision detection performs decently across a wide range of inputs. Highly complex and concave objects cause problems, but this is to be expected, given the nature of

★★★★★ excellent
★★★★ very good
★★★ average
★★ disappointing
★ don't bother

the beast. Reactor's fluids, soft body dynamics, and cloth are not completely convincing, but on par with the simulation experienced in today's games. Reactor's weak support for highly articulated structures is somewhat disappointing, but I expect that this will be improved in the near future. For now your primary characters will need to be physically agnostic — affecting the secondary dynamics of a scene but themselves oblivious to their kinetic interactions.

Anyone familiar with incorporating dynamics into a game knows the value of a real-time preview. No matter how well you understand physics and animation, you will need to iterate to get what you want. If you do not completely understand every physical parameter and setting required to add dynamics to a scene, trial and error can be an invaluable learning tool.



In summary, Reactor is a step in the right direction. It combines a number of powerful physics technologies into a somewhat unified, conditionally stable, moderately intuitive package — slightly less than what you would ultimately wish for, but a lot more than anyone else has to offer. If you are a game developer licensing the Havok physics engine, Reactor is a must-have. If you have your own physics system, Reactor might be a highly economical path for getting physical attributes into your game.

★★★★ | REACTOR
developed by Havok for Discreet
www.discreet.com

RIGHT HEMISPHERE'S DEEP PAINT 3D WITH TEXTURE WEAPONS

by steve theodore

Deep Paint 3D from New Zealand–based developer Right Hemisphere is a full-featured 3D painting application that allows users to paint directly onto 3D models. The program exchanges models with 3DS Max, Maya, and Softimage and supports other applications through .3DS or .LWO files.

3D painting is remarkably similar to the familiar 2D painting process. Performance in 3D painting is surprisingly good, nearly real-time even on mid-range machines. Tablet feedback is excellent, and feels smoother than Photoshop's. Models in the 3D window are rendered with not only color but also bump specularity and self-illumination maps. Each map channel can contain an arbitrary number of layers, allowing experimentation and the isolation of details. Unfortunately, the layers offer only simple or multiplicative compositing — Photoshop users will miss modes such as additive and "soft light."

Deep Paint 3D borrows liberally from Photoshop in its selection tools, "quick mask" friskets, filters, and even hotkey assignments. Despite these influences, however, the program really resembles Corel Painter in spirit. Its primary tool is an extremely configurable "freehand" brush that can render excellent imitations of traditional media (chalk, airbrush, and the like) or be tweaked for interesting technical effects. The brush mechanism is also used for most image-processing tasks — functions such as dodging, desaturating, and sharpening are performed with special brushes rather than filters. Image-based brushes aid in rubber stamping and cloning textures.

The "everything-is-a-brush" structure is powerful, but not friendly. While Deep Paint 3D includes numerous preset brushes, it's taxing to wade through dozens of names to find a brush you need. Apart from a nice method of interactively resizing a brush without an options dialog, the interface does a mediocre job of organizing the program's complex tools.

Quality painting in Deep Paint 3D depends on the underlying UV mapping of the model; if adjacent mapping areas have mismatched texel densities or orientations, visible seams and erratic brushwork may be hard to avoid. The Texture Weapons add-on is an excellent tool for modelers whose 3D packages have poor UV-mapping abilities.

Texture Weapons can apply UV coordinates to an object so that texel density is equal everywhere. The practical result is the elimination of projection smears and pinch points. This process produces lots of discrete patches in UV space, effectively trading more seams for fewer texture smears. This fragmented mapping can be inefficient; a disjointed map may use fewer than half the pixels in a given bitmap.

Texture Weapons allows you to tweak the trade-off between texture distortion and fragmentation, and provides strong UV editing tools to combat inefficiency and seams. UV coordinates can be scaled and welded to minimize seams and make more efficient use of texture space. A unique addition is the ability to preserve a texture after editing UVs, so that a reorganized mapping does not demand repainting.

Texture Weapons also adds "Projection Paint" mode, where painting operations are done in screen space, independent of the model's UV maps. This means brushes no longer change resolution as the texel density of the model changes. Even better, seams are disguised by the downsampling of the projected image. Moreover, it is possible to paint very effective subpixel detail simply by painting from a zoomed view. Projection paint is less interactive — entering and leaving the mode can take upwards of 30 seconds for a complex model — but provides a high-quality solution to the common annoyances of 3D painting.

The Deep Paint 3D and Texture Weapons combination has some rough edges. The interface is far from self-explanatory and some of its most interesting functionality is buried deep within options dialogs. Small details are skimped — there are no tool tips, for example, and the hotkeys are not remappable. Live import and export communications with Max and Photoshop is occasionally flaky. Despite these imperfections, Deep Paint 3D is a solid tool which performs an important function. For packages with weak UV tools, particularly Max and Lightwave, the Texure Weapons add-on is invaluable. For anyone who is interested in sophisticated texturing, Deep Paint 3D demands a serious trial.

★★★★
DEEP PAINT 3D WITH TEXTURE WEAPONS
Right Hemisphere
www.righthemisphere.com

mark deloura | TALKING TO PEOPLE WHO MAKE A DIFFERENCE

George Sanger
The Fat Man Cometh

George Sanger, also known as The Fat Man, is absolutely unmistakable. From his Texan hat and Nudie suits to the fat sounds he created for legendary titles such as WING COMMANDER and THE 7TH GUEST, George is an industry original. In his copious spare time, he organizes Project BBQ, an audio industry brainiac camp-out, and GamePlayMusic, a flat-rate music library for games.

Game Developer. Are you really working on slot machines?

George Sanger. The people in the slot machine business are in a very lucky place. They can tell when something is good, because it makes money. They've been able to cleanly measure that good sound makes more money. Because of that, I'm designing a Fat Sound speaker system for slot machines, and redesigning sounds for a whole bunch of slot machines. Some of them still only have the brain power to play four sounds, so I'm replacing four clanks with four different really nice clanks. On the newer games, we've got whole new elaborate systems in there. It's PC work, but we know what PCs are in there. It's not a moving target, and that's really pleasant. So I'm working on speakers, I'm working on sounds, and I'm working on sound systems. The challenges include making audio that's appropriate to the environment that's there — in other words, it's not going to sound like an arcade. My goal is to make a very pleasant place for a human being to be for a long time. That's what it's about, doing something positive when you have a chance to. There are so many opportunities to do remarkable things right now with audio. And so many of those opportunities are still open and unexplored — it's a very exciting time.

ABOVE. George Sanger, The Fat Man.

GD. What kinds of opportunities?

GS. When I gave my first talk at the GDC in 1992, it was possible to make a game soundtrack as good as anything on TV or in the movies. It still is, but people aren't thinking, "We could win a Grammy." Have we done anything that's better than the movies yet? I don't know. But that should be our slogan as an industry: "Not as good as a movie, better than a movie." Some of the smartest people in audio don't feel that interactive music can be done to the level of satisfactory art that linear music or visuals can be done. And the reason is that audio is all about timing, and you don't have control over the timing. But if we consider that there is a legitimate art form called interactive music, it follows that it would be like linear music, and it would be satisfactory. But it's influenced by what the user does, and how the user experiences it, so that regardless of what the user does, it's always a satisfactory experience. This is really similar to the phrase I heard when I took a sculpture class: "effective when viewed from any angle." Whatever interactive music is, it would be to linear music what sculpture is to painting.

GD. How is GamePlayMusic going?

GS. It's remarkable, GamePlayMusic has not gotten a single response from anyone. And there's a mystery for you. I mean, from musicians it has. But I have this fabulous resource which is about two feet of CDs, full of remarkable music from people all around the world, all sorted by categories like battle, medieval, high-tension, low-tension, walking around in mystery, and it's all keyed right to our industry. It would just blow your mind to hear some of this audio.

GD. Do you think that audio designers are reluctant to purchase music that isn't specifically designed for their title?

GS. Absolutely. And to them I say, that's not your problem. You could certainly write a small number of tunes to go specifically with your title. But people don't complain about computer game music not having enough unique melodies. What people do complain about is hearing the same music over and over again. We're in business here to do something aesthetically pleasing, we're here to entertain. And we're screwing up in the audio department. Seriously, we've gotten the message: repetition is the problem in game audio. And somehow what we hear is: it's those tiny speakers. Well, it's not! Every time you think you hear someone say, "It's those tiny speakers" — I don't know who hypnotized us into translating one as the other, but televisions with little speakers do fine!

GD. If you're an audio guy and want to break into doing interactive audio, where would you start?

GS. It would behoove them, to get a start, to go to the GDC, just to get a feel for it. I've been telling my son, who really wants to make bamboo windchimes and sell them door to door, don't sell anything until you've done it about 10 times. So score 10 games, for nothing, for yourself. Don't put them in a game, just score them. And see if you like that. If you do, well you've just got yourself a hell of a great hobby, and now your life is bearable. And if you can't stand doing it, then get out of the business before you get in it.

The answer is, "Just do it, for godsakes just do it, and don't think about the money." Put the energy into it that it deserves, and make it sound really good to you. And then, that's all, and see if success comes. I have a feeling it will if you're sincere.

Return

Last Call at the House of Blues

Is enough ever enough? When are computers fast enough for games? Are we nearing the time when 3D graphics cards have so much fill rate and polygon throughput that we have all the triangles we need? Questions like these come up from time to time. Usually the question comes up in the public relations blurb for the latest hardware release, hot game, or game engine. You get quotes like, "We now have all the polygons we could ever need," or, "Our engine can push so many polygons that our artists have trouble finding places to add any more."

There are definitely two sides to the argument. Games should be about the gameplay and not necessarily the look of the objects in it. It should be obvious that simply having more polygons to work with doesn't make a game more fun. PONG is still a pretty fun game with only a few triangles making up the display. Have an ancient Atari 2600 lying around? Spark it up at your next party and many people will still want to play COMBAT. Certainly we all have played quite a few games full of cutting-edge visuals and massive polygon counts that just weren't fun at all.

However, games that look better can certainly be more immersive. Reality is much easier to suspend in a world full of rich details. We also can't ignore the fact that consumers who buy these new consoles or latest computers with high-end graphics accelerators are expecting more realistic and detailed graphics with each new game released. The hardware manufacturers certainly don't believe we have enough power yet. It is in their financial interests to keep the demand for more and more computing power high. That way they can continue to sell more and more powerful hardware every quarter.

It is true that production issues make creating this incredible amount of art assets difficult. Increasingly we are also starting to hit memory storage, loading delay, and bus bandwidth issues. But the need for more computing power still exists. We don't just need the power to create more detailed models. With the power available now, artists can create pretty compelling characters and environments. Sure, we could add a few more polygons here and there, round out some curves, add some detail to the gothic archways in our dungeons — but that is not where the real problem lies. What game worlds really lack is "stuff."

Many games have very detailed architecture with realistic shadows, but the environments are totally lacking in clutter. Just modeling the amount of clutter on my desk where I am typing this column now would easily bring the fastest available system to its knees. Sure, the stuff may not be interesting to the gameplay, though I obviously think I need it on my desk or I would throw it away, right? But, it is that level of detail that takes the game environments to the next level of authenticity. Maybe it is time to start thinking about having virtual set dressers that walk through a level, strategically dispersing clutter and placing commercial products for endorsement deals.

Game worlds also lack people. How many nightclub sets have you walked through in a 3D game? There are always a couple of people in there, one of whom you can usually talk to. Now, I know why that is a problem. People take up a ton of polygons. They are hard to make look good and even more difficult to make move well. Creating animation for characters is one of the most difficult and costly parts of production. That is why you don't see too many people walking around our interactive worlds, and even when you do, they are pretty lifeless. But people are the critical parts needed to make most games more credible. The world is full of people and we as players are used to that, so until I can walk into a dimly lit, smoky bar, packed shoulder-to-shoulder with people and a blues band on stage, I don't think we have nearly sufficient horsepower to make things interesting.

Inside the Smoky Room

I am very confident that the makers of computing hardware will keep improving the machines until we are fully capable of creating the smoky bar scene I just described. The hardware isn't really a barrier. Just throw some time and some smart hardware

JEFF LANDER | *Visions of a smoked-filled speakeasy where men are mysterious and dames even more so obscure the rays from the summer's sun. Help Jeff lift the fog from his world at jeffl@darwin3d.com.*

guys in fierce competition, and those issues will be addressed. Creating the content to put on this hardware is the tricky bit. For example, look at the characters that are in the bar. Someone is going to have to build all those characters and bring them to life. For the characters that are integral to the story and action of the game, this is an expensive but very necessary task. Once the graphics and animation systems are in place, it is just a matter of creating the mountains of content for those characters.

But what about all the dozens of other people that we need to populate our bar? Each of them needs to have a distinctive personality, unique look, and characteristic movement. Who is going to create all of them? Probably not your lead artists if you ever want to get the project finished.

In the case of movie production, we would just have a casting call for extras. A production assistant would go out and get a bunch of people who look and act like ordinary people and can fill out the scene. In interactive entertainment, however, we do not currently have the equivalent of extras that we can bring in for the production.

Certainly, we can make some character meshes and a set of generic motions that can be applied to all of them. We could trigger these different animations based on some sort of artificial intelligence system like a finite state machine. This kind of thing has been done before. There has even been some licensable technology to provide this level of character function. An animation system like this can be effective. However, these extras have a pretty limited repertoire. If they are required to do any kind of new behavior, they may not be up to the task. For example, if you walk up to an extra and trip him or hit him on the back of the head with a chair, unless that is in the character's animation set, nothing will happen.

What we really want are intelligent physical objects. In a car race game, we can set up rules for the physics of the world and the cars. Then we can just drop a bunch of these cars in the world and let them go, driven by simple AI that controls the motion of the cars. We can teach the cars special moves like navigating through traffic and parking so they can get around. However, since the game has a basis in physical simulation, when the unexpected happens, like some cars run into each other, the result is something that is both unique and believable.

As you probably know, making a physical simulation for a bunch of cars is a much easier problem than simulating people walking around a room. In fact, it is not just a single problem; it is a whole series of difficult problems. The task of simulating a human in a game environment seemed so daunting to me until a few months ago. I thought it was a goal that we should shelve for a few years until the hardware became powerful enough to handle environments full of passively animating characters. Then we could tackle how to automate them more believably.

A few things have happened that have really changed my mind about this. For one thing, we have the knowledge and ability to create pretty interesting characters. Technologies such as facial animation, motion blending, secondary motion, and on-demand inverse kinematics are well understood and pretty easy on modern game hardware. Yet characters in games are only slowly getting better. Talking to developers, it seems that production issues, not technology, is what is really holding them back. Knowing how to animate a face is one thing, but creating streams of animation to actually perform the action is really time consuming. Simulation could help with this issue.

Second, I have learned that we don't need to grab for the final prize of fully simulated characters in order to achieve some major advancement in character interaction. I, as well as others, have combined some fairly simple simulation systems with traditional blended animation. This has been moderately successful. The characters seem much more alive, since they can, at least to a limited extent, react directly to the circumstances around them. With this in mind, I am going to take up the gauntlet of physically simulated characters and see what I can learn.

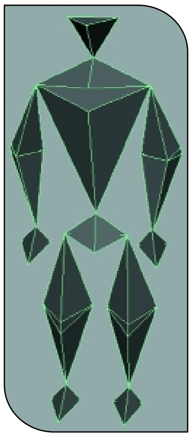


FIGURE 1. A 16-link rigid-body man.

Take a Seat at the Bar

The problem of physically simulating a living creature can be broken up into two major pieces. The first part is simulating the physical kinematics of the character. That is, simulating how the body moves through the connection of the various limbs and body parts. The second part is controlling the body parts to make the character move. This "controller" problem is very difficult. You can equate it with teaching a baby to learn to walk. Simply staggering along without the legs wobbling too much is a pretty big accomplishment. Learning how to dance like Fred and Ginger is going to take awhile.

For now, let me stick with the kinematic part of the problem. A human being is composed of linked rigid bodies that are connected in a hierarchy. Anyone who has created a character in a modeling package and animated it by using a bone system should be familiar with the idea. For example, in a simple human character, at a bare minimum I probably want to use 16 connected rigid links: head, neck, upper body, lower body, upper arms, lower arms, hands, upper legs, lower legs, and feet. You can see a graphic representation of this linked system in Figure 1.

Each link is connected together at a joint. In the case of my character, there are two types of joints. Some joints have one degree of freedom and can bend only along one axis, like the elbow or knee. Other joints can move about all three rotational axes, like the shoulder or hip. Obviously, joints are not entirely unlimited even in their free direction of motion. A knee cannot bend around 360 degrees, for example. Restrictions on joint rotation (called joint limits or degree-of-freedom restrictions) are used to keep the joint from over-rotating. I will get back to that issue in a moment.

The mathematics of rigid body motion, where the objects are linked by joints, is called constrained rigid body dynamics. A very good introduction to the topic can be found in Chris Hecker's two-part article in *Game Developer* ("How to Simulate a Ponytail," March and April 2000). For now, however, I am going to ignore the fact that the links are rigid.

In previous columns, I have had a lot of luck simulating soft body objects by using point masses connected by springs. I have even been able to extract a rigid link orientation from an object composed of these connected point masses. A rigid body can be thought of as a series of point masses separated by constant distances. So, if we are able to maintain the constant distance between particles, we could simulate a rigid body using simple particle physics.

At the Game Developers Conference this year, Thomas Jakobsen from IO Interactive presented a system that he used in the game HITMAN: CODENAME 47 to do just this. The developers connected points representing parts of the body into a rigid hierarchy. By enforcing various constraints to make the joints bend correctly, they were able to simulate the degrees of freedom of a simple human character. They then used the system to simulate characters that could collide with the world and fall to the ground realistically.

Four point-mass particles are connected by rigid links in the form of a tetrahedron, creating a stable rigid body that will not collapse on itself. For the body to be truly rigid, I need to make sure that the distances between the points are kept constant. When creating a soft body that can squish slightly, the separation distance can be maintained with springs. In order to keep the objects fairly rigid, the springs will need to be stiff. If not, this can lead to numeric instability and crash your simulator as I discussed in my column on cloth simulation ("Devil in the Blue Faceted Dress: Real-Time Cloth Animation," May 1999). Better numerical integration techniques can address this problem, but to maintain exact distance between points, it may be necessary to move the points after integration in a fixup stage as described by Xavier Provot in his paper on cloth animation (see For More Information). In HITMAN, Jakobsen used multiple fixup passes instead of springs to maintain the rigid links between points. If the points are moved in a fixup pass, it may be necessary to correct the point velocity to insure the integration scheme doesn't break. Jakobsen avoided this by using a velocity-less integrator like the Verlet integrator described in his paper. In my experience, either method works well.

To link the rigid bodies, a single point of the tetrahedron can be constrained to stay in place. This allows the rigid body to rotate about that point in three degrees of freedom. If I then constrain another point of the object to stay in place, that removes two degrees of freedom, leaving an object with one degree of freedom. With these two types of constraints, I can start building my character.

How About Some Elbow Room?

Let me start building the object with a simple linked chain for the arm. The arm is made of two connected tetrahedrons, as you can see in Figure 2.

The position of point *b* represents the shoulder joint and should be pinned in place. The two links are connected at edge *c*, which represents the elbow joint of the arm. While the lower arm link can rotate around edge *c*, it is not free to rotate in any other direction. But, it is free to rotate completely around *c*. That means the angle *a* can vary from 0 to 360 degrees. That is obviously something that would be physically impossible with a real elbow. A real human elbow can bend up to about 145 degrees. This can be enforced either by checking the angle between the joints with the dot product or by limiting the size of the gap between the joints. The over-rotated joint can then be pulled back either with a spring or by using a rigid constraint. The benefit of a spring is that the joint will tend to return to a comfortable "rest" position. Large over-rotations can also then appear to cause a hyperextension or "dislocated" joint.

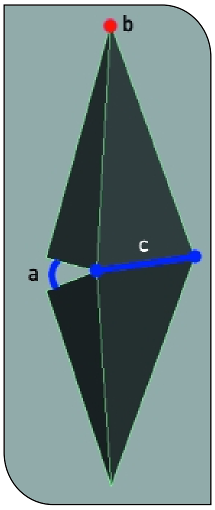


FIGURE 2. A simple arm chain.

Constraints can also be used to keep joints from colliding with each other. Since I don't want the arm to pass through the body, space between the arms and body can be maintained with minimum distance constraints.

He Slid Under the Table

By using these techniques, the physical representation of a character can be created and all of the joint constraints added. This is done automatically, creating the different forms of links based on the degrees of freedom and range of motion for each joint. Once that is done, I can stick the character in the simulator and let it run. Since the character is just composed of simple particles connected by simple constraint functions, the processing time is minimal. The routines are also ideally suited for parallel processing.

However, the character doesn't do a whole lot right now. If you apply forces to the links, the character will move in a physically believable way. Apply gravity and collision detection with complex environmental objects and the character could be made to crash into objects and fall over in a somewhat realistic manner. For HITMAN, this feature was enough to make the system worthwhile.

But, to make things more interesting, we need to integrate these passive dynamic effects with more active systems. I really want to create characters that can balance and move on their own, or take a hit and then react with plausible and appropriate motion sequences. That will just have to wait till next time. It's time for me to order another drink.

FOR MORE INFORMATION

Jakobsen, Thomas. "Advanced Character Physics." *2001 Game Developers Conference Proceedings*. pp. 383–401.

Provot, Xavier. "Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior." *Graphics Interface*, 1995. pp. 147–155.

Return

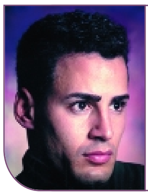
Efficient UV Mapping of Complex Models

There you are at your computer, tumbling a nicely designed 3D polygonal model that needs texturing. Maybe it's a highly detailed architectural structure or some organic alien plant life created in your favorite 3D package. The assignment now is to texture-map this intricate model for a real-time game and then create the actual custom images you will assign to it. Aesthetically speaking, your texture-mapping work should not only be well detailed and designed but also appear seamless on your model and have a relatively even distribution of pixel resolution throughout your model. Technically speaking, your mapping should make efficient use of your texture sheet by using the maximum possible area of the texture space. Sound easy?

The process for assigning mapping coordinates is quite the same whether you are creating 3D geometry for a PC title, a console title, or a web-based game. The actual steps you take for setting up and mapping texture coordinates onto complex geometry are equally laborious for each of these platforms. The focus here is to get you through this less creative process efficiently and with great accuracy so that you can have more fun creating the bitmap images afterward. The more intuitive and interactive this creative process becomes, the better control you'll have and the more convincing your final result will be. In your efforts, you may find a plug-in or two that will help expedite a few steps in this process, but the basic steps are always the same. The "make it so" one-click feature that creates the perfect mapping hasn't been written yet for any 3D program. Until that happens, here are some proven basic steps you can take that will help you prepare your models for skinning and produce great mapping results every time. (For the rest of my column, I will assume you are experienced with 3DS Max 3 and know your way around Adobe Photoshop.)

Analyze Your Model's Geometry

If you are the original modeler of the 3D object you are about to texture-map, then you're already intimately familiar with every detail and polygon that makes up this geometry. If another modeler handed off the model for you to texture, take the time to analyze the design and look for ways you can begin subdividing the



TITO PAGÁN | *Tito is a seasoned 3D artist/animator working at WildTangent and teaching at DigiPen in Seattle. His e-mail address is tpagan@w-link.net, or visit his web site at www.titopagan.com.*

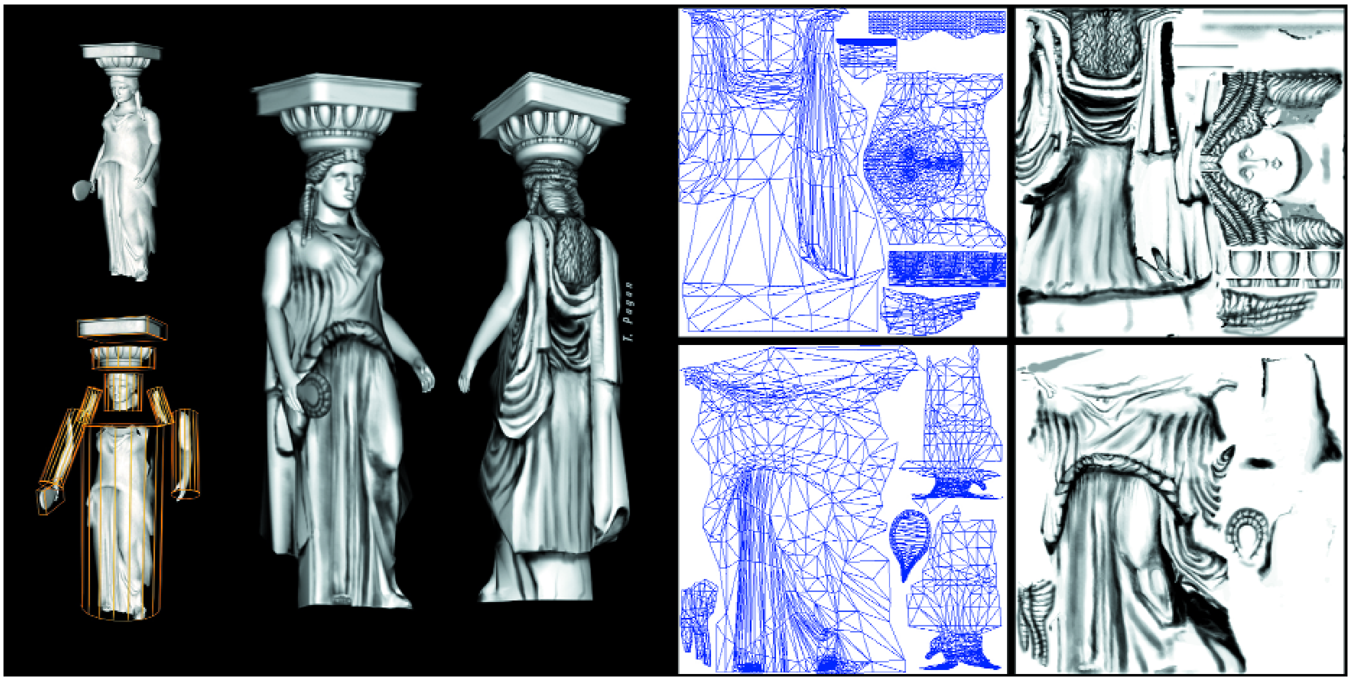


FIGURE 1. A decorative structural column (14,419 polygons) with precise UV mapping.

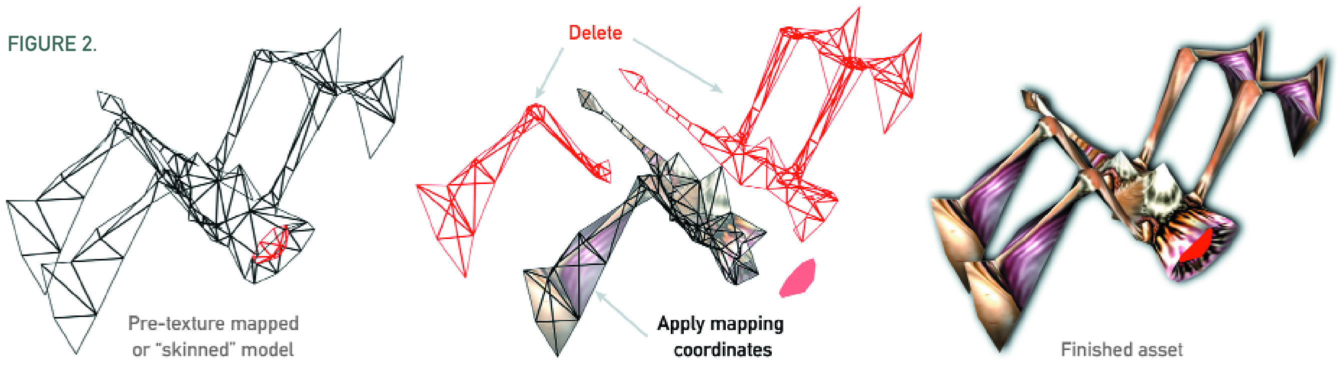
entire model into smaller groups of more primitive-like shapes (see Figure 1). Doing so will make your next step easier by breaking down your model into manageable parts to which you can assign separate mapping coordinates. For example, a spider's body can be thought of as narrow cylinders attached to a couple of spheres or a capsule-like shape. Look for elements in the design of the geometry that are identical. A spider's eight legs can be the same identical pieces without compromising much of the design. This is probably the way it was created to begin with — one leg was created and then cloned seven more times for a full set.

Subdivide and Consolidate

For this step you will dissect and delete parts of the model. If you work in a production environment with other contributing artists like I do, this may feel uncomfortable at first because you may be taking apart someone else's model. This is a necessary and practical step that can't be avoided. If the identical or repeating mesh detail is to have the same bitmap assigned to it, there is no point in defining mapping coordinates for this detail twice. Nor would it be efficient to use valuable texture space to represent the same texture detail on the model's similar geometry more than once. This conservation of time and consolidation of texture space can be handled in two ways.

The first approach you can take is to subdivide your model and delete anything that is represented in your model more than once (see Figure 2). This is the approach I often use. Focus on doing a good job applying mapping coordinates to only one of these similar parts. Later you will duplicate and attach the newly mapped parts back to the model in their original positions. Again, fewer pieces of geometry representing your entire model will equate to fewer polygons, less work in mapping these sub-parts of your model, and therefore more texture space real estate you can devote to these fewer parts.

FIGURE 2.



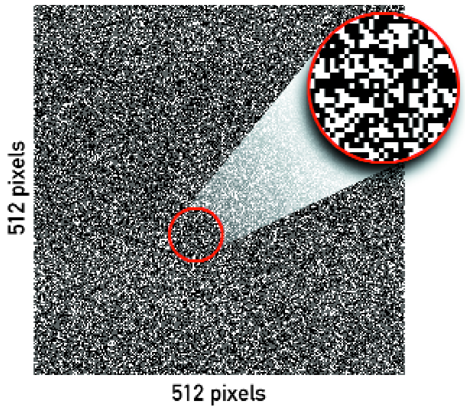
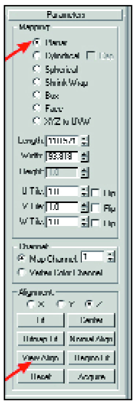
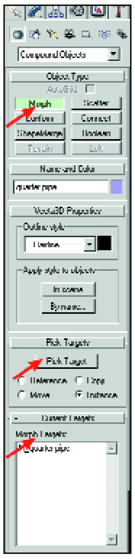
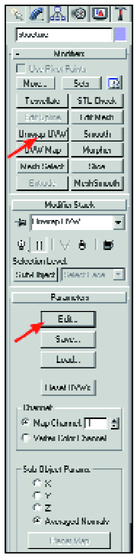


FIGURE 3 (above). Create a temporary bitmap and assign it to your model. FIGURES 4, 5, and 6 (below, left to right). The editing modes of the UVW Map and Unwrap UVW modifiers.







The second approach is to leave your mesh object intact and do a face-level subobject selection as many times as you need to and apply separate mapping coordinates for each selection. I don't recommend this approach because it is more work than it's worth, takes more time to execute, and doesn't yield better results.

Assign a Texture

You are now at a good place to assign a bitmap to your model. Create a texture file using your final asset name and place it in the directory for your game's final bitmap assets. Assign it to the model from within Max's Material Editor. This is usually a .BMP or .JPG image file that is 512×512 pixels in size or smaller. I like using the highest possible resolution for texture images that the game will support while I create them. This enables me to design and evaluate my textures at their most optimal level. It also gives me the flexibility of easily downsampling this size later to 256×256 or 128×128. This will depend on resolution limitations imposed by low-end graphics cards or file optimization issues that often arise toward the end of many projects. Another common practice is to create a "test purpose" temporary bitmap that has a consistent pattern throughout and assign it to your model. A checker pattern of small black and white squares is what I have found works well (see Figure 3). Other artists use numbers and colored squares in their test texture.

Applying UVs

Applying your initial mapping coordinates is a relatively simple process. 3DS Max offers many options for specifying how bitmaps are projected onto the surface of an object. Mapping coordinate types such as Planar, Cylindrical, or Spherical can be found using the UVW Map modifier (see Figure 4). Here again is a step in the process that can be handled in several ways. A high-production 3D artist will typically develop proficiency in Max for at least two or more approaches. I will discuss two of my personal favorites.

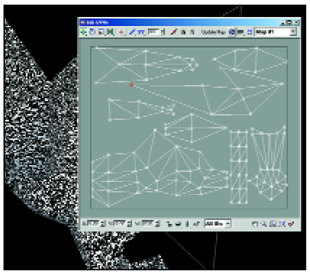
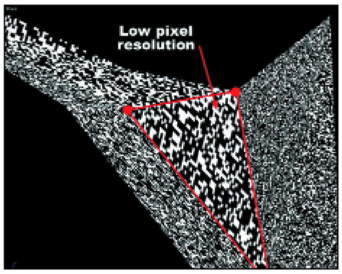
The first approach is to take your ready-to-map pieces of geometry and break them down into a "flat pattern" to which you can then apply a Planar-type mapping coordinate. This approach is like breaking down a simple cardboard box. A more complicated cardboard box, like a closet shoe storage box with cubbyholes, would of course present a bigger challenge to flatten out, but it can still be done. When you think about it, even an automobile's exterior body once started out as a series of flat-patterned sheets of metal that were formed and welded into a unique shape. This idea of reverse engineering the construction of your model's geometry requires you to know your way around Max's vertex manipulation and translation tools comfortably. You can find third-party plug-ins to help you through this process. Just remember always to clone a copy of this shape before you deform it into something suitable for mapping, because you will need it to morph (a compound object type for creating geometry) your newly deformed object back to its original shape. Use the copy as a morph target after you are done with your map assignments (see Figure 5). Avoid deleting or creating new vertices in either copy of this object. This is important to remember if you want to guarantee you'll get back to the original target shape during the simple morphing process. You can temporarily hide this copy to get it out of your way while you work on the other.

A second common approach to applying initial mapping coordinates is to leave your object's mesh intact and select a small group of faces you wish to planar-map. Make sure that these selected polygons all face the camera as if you were looking down their normals. This puts your selected polygons perpendicular to your camera. Assign a UVW Map modifier and set the alignment to View Align, thus reorienting the mapping icon to face the active viewport, which is the camera. This will be good enough for a first pass at applying mapping coordinates for these faces.

You could have easily rotated your object to put these selected faces perpendicular to an active orthographic viewport. Oftentimes, however, translating an object in world space presents problems with alignment to other objects. It can also throw off orientation during integration into your game. I don't recommend it.

Evaluate Your 2D Image Space

The Unwrap UVW modifier is used to assign planar maps to subobject selections of polygons and to edit the UVW coordinates of these selections (see Figure 6). I strongly suggest that you become very familiar with the Unwrap UVW modifier. We will use it as a UVW coordinate editor to unwrap and edit the existing UVW coordinates which were created earlier using the

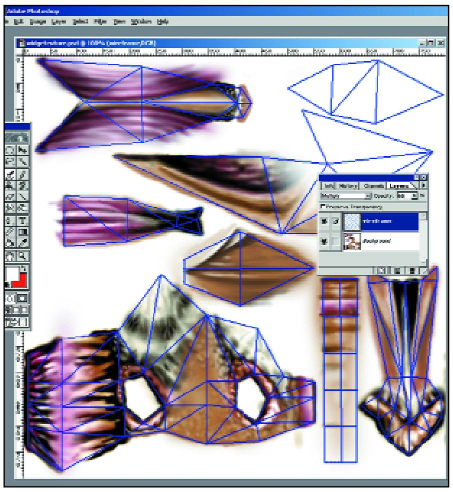


FIGURE 7 (above left). View your textures in the UVW window that displays the faces of your entire model. FIGURE 8 (above right). Scale your UVW vertices for consistent pixel resolution. FIGURE 9 (left). A dark line wireframe reference image with its layer-blending mode set to Multiply and a low Opacity setting.

UVW Map modifier. When the Unwrap UVW modifier is applied, it takes the current UVWs applied to the object and stores them in the modifier. If the incoming data on the modifier stack (from bottom of the stack upwards) is a face-level subobject selection, then only the UVWs for the previously selected faces are brought into this modifier. These same selected faces will display in the UVW window as UVW faces and UVW vertices overlaying a 2D image space of the map. (Technically, I should refer to bitmap coordinates as "UVs" instead of "UVWs" because the "W" represents the axis that is generally only used for procedural maps.)

You will want to scale and adjust your UVW vertices to ensure you are getting a consistent pixel resolution and distribution throughout your model. As you move the vertices around, you will see the texture move about the surface of your model as it updates in your viewport (see Figure 7). I suggest you turn on Texture Correction at this time by right-clicking the label of that viewport. Look at the checkered squares on the surface of your model. If areas appear to have rectangular checkers instead of square checkers then you have stretching going on in the direction of the longer side of your black and white rectangle. To correct this, select and scale the UVW vertices that define those faces in Unwrap UVW using the Scale Vertical or Scale Horizontal tools. If the checkers in some areas appear to be larger or smaller than most (see Figure 8), adjust in Unwrap UVWs using the uniform Scale tool until they better resemble the other checkers.

Don't be overly concerned about trying to use all of your 2D image space. Depending on your texture detail, contrast, and complexity, maintaining a consistent pixel resolution and clear surface detail may be more important than using up every possible area in your 2D image space. With time and experience you will get better at deciding which is more important as you adjust your UVW lattice. When you are done with all of your mapping you can weld your vertices back to quickly re-create the single object with which you originally started.

Create Your Bitmap Iteratively

Since a correctly named texture is already referenced to your model and the mapping coordinates are also properly applied, you are now ready to "improve" your existing bitmap. If you're like many 3D artists I know, the excitement is building up knowing the fun part is only moments away. You've just worked hard to prepare your model and you're about to slap down some cool colors and rich detail that will take this content to the next level of believability. I recommend that you do a couple of things before starting to make this last iterative step feel more interactive. Open and view your texture space in the UVW window that displays the UVW faces of your entire model. Take a screenshot to capture the 2D image space of the map that shows all of the UVW lattice lines. This will put a copy of what you see on your screen into your system's clipboard. Now launch Photoshop, if you don't already have it open, and create a new image by clicking on Control-N. Hit Enter when the options box comes up to accept the new image file with the default dimensions. Paste your clipboard image into it by clicking Control-V. Now you have a wireframe print of your UVW lattice for your mapped model as a reference. Crop your image to include only the 2D image space area in the image. Resize this image to 512×512 pixels using Image Size in Photoshop, making certain that the Constrain Proportions option is not checked. I like having a dark line wireframe reference image in a top layer with its layer-blending mode set to Multiply and a low Opacity setting (see Figure 9).

Finally, to help automate the iterative process of saving an updated copy of your image over the existing bitmap referenced in Max's Material Editor, we'll create a Photoshop Action. By doing so you can duplicate your image, hide or delete your wireframe reference layer, merge your other layers, save a copy of this image file over your existing one, and close the file, all with a single click of a button. You can now jump back into Max by hitting Alt-Tab. With the Material Editor open and your Bitmap Parameters>Reload button showing for your objects material, you're a click away from seeing how your new texture is shaping up, all in the context of your model or scene. Can you want more than that without the expense or learning curve of other third-party software?

These steps are all you will need to know to improve your skinning process and get up to speed with industry veterans. By eliminating tons of guesswork during the mapping stage, you can save yourself lots of time and headaches. Good art alone requires enough of your time already — these basic steps should buy you more time to create just that.

Game AI: The State of the Industry 2000–2001

It's Not Just Art, It's Engineering

If last year's Game Developers Conference made it abundantly clear that game AI and AI developers had finally "made it" and were being taken seriously throughout the design and production process, the 2001 conference took things up one more notch. Developers are turning to tried and true engineering techniques to build their AI, and it shows in the quality of their products. No longer is AI jury-rigged to meet the needs of the schedule; developers are finding themselves with the time to actually design their systems for maximum flexibility and playability. "Fuzzy" technologies have fallen by the wayside in favor of solid, well-thought-out, easy-to-modify designs — even Dr. Marvin Minsky's packed keynote address at the GDC noted this trend.

Nearly every engineer I spoke to cited one game as inspiration for taking the time to do AI design right: Will Wright's masterpiece, THE SIMS. The success and flexibility of THE SIMS' design is easy to see — the game continues to sell at the top of the charts more than a year after its release, and whole legions of add-on packs and user-built objects are available to extend and strengthen the lifetime of the game. By taking the time to carefully craft his "smart terrain" design, in which the terrain and objects within the game broadcast their ability to satisfy the needs and desires of the Sims within the game, Wright was able to lay the foundation for an AI system that was extensible in ways he probably never imagined. Users in turn have responded by building and releasing hundreds of objects; the most innovative is probably the radio object released as a publicity stunt that tunes in to a streaming broadcast of that station over the Internet.

The influence of THE SIMS was evident throughout AI designs described at the conference. Developers have found that having a solid design makes everything much simpler if done correctly. It's not a panacea of course. As the SHADOW WATCH team noted, their work is still "60 percent design and 40 percent plugging holes." But the industry as a whole is a fair sight farther along than it was a few years ago, and the success of the thorough AI design in THE SIMS will only serve to benefit game players in the future as developers wrestle with the problems of multiple platforms and ever-shortening schedules.

Resources Revisited

This year's roundtables showed that the resource availability trend for developers continues to hold steady from years past (see Figure 1). Nearly 94 percent of developers attending the roundtables reported that they have at least one dedicated AI developer on their current or most recent project, and several have two or more. One developer, Quicksilver, boasted a total of three AI developers for their upcoming MASTERS OF ORION 3 (not unreasonable, given the immense scope that the game promises to deliver).

Developers also continued to report that they have more than enough CPU time available. No developer present thought that they had too few CPU cycles to do the job, and some developers were downright embarrassed at the riches available to them. As in years past this trend in part represents the enormous growth in power of the CPUs available in systems today — there's just more horsepower to be had. Developers continue to

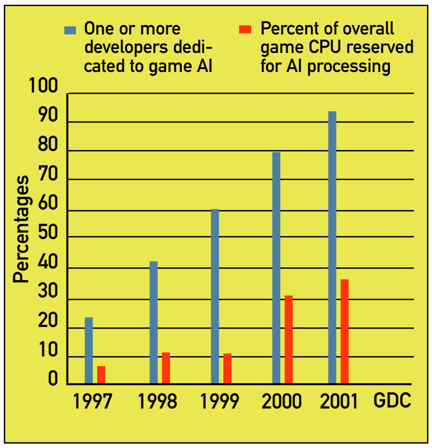


Figure 1. The 2001 AI resource picture.

STEVEN WOODCOCK | *Steve's background in game AI comes from 18 years of ballistic missile defense work building massive real-time war games and simulators. He did a stint in the consumer arena, then returned to the defense world to help develop the AI for the national missile defense system. He maintains a web page dedicated to game AI at www.gameai.com and is the author of a number of papers and publications on the subject. He now pursues game AI through a variety of contract work and served as contributor to and technical editor for several books in the field, most recently the* Game Programming Gems *series. Steve lives in gorgeous Colorado Springs at the foot of Pikes Peak with his lovely wyfe Colleen and an indeterminate number of pet ferrets. His hobbies include hiking, shooting, writing, and working on old GMC trucks (go figure). He can be reached at ferretman@gameai.com.*

Illustration by Claudia Newell

ABOVE. SHADOW WATCH'S AI is both analytical and extensible. RIGHT. The AGE OF EMPIRES series continues to produce modular, well-engineered games with strong foundations in design.

credit this abundance to the powerful graphics cards which are now fairly common in players' systems, but the fact that all of this extra CPU power isn't just being spent on eye candy is telling.

Tools of the Trade: Trends of the Past Year

The single most striking development evident in the past year is the arrival of true engineering techniques and discipline to the AI problem. Not to say that many developers are doing anything different than they've ever done — there are many excellent developers that have always taken the time to design their AI engines in the most efficient fashion possible. But it was evident at this year's GDC by the success of modular, well-engineered games such as THE SIMS and the AGE OF EMPIRES series that more developers than ever before are being inspired to focus on their designs before writing a single piece of code.

Many developers have been taking their design cues from studying the way that nature has built organisms to handle their environments. Artificial Life (A-Life) techniques continue to gain popularity for the simple reason that they work. A-Life seeks to emulate the behavior of real-world organisms by allowing developers to break complex actions down into simpler, smaller pieces. Combining the execution of these many little actions leads to a phenomenon called emergent behavior. When tied to the motivations and needs of individual AIs (for example, "I really want to capture that city over there"), the interactions that occur between the low-level, explicitly coded behaviors and the higher-level needs of each AI result in the emergence of higher-level, more sophisticated, more "human" behaviors. This in turn makes the AI look smarter and appear more motivated than scripted, and more decisive than reactive.

Several of the presentations at this year's GDC reflected this continued interest in A-Life techniques for games. Demetri Terzopoulos, John Funge, Bruce Blumberg, and Craig Reynolds collaborated on a full-day tutorial titled "Artificial Intelligence for Computer Games" which focused on all kinds of A-Life and biological techniques. Terzopoulos focused on building artificial animals with artificial sensory systems, while Blumberg's portion of the talk discussed learning in the animal world using dogs as a model, and how these techniques might be adapted to non-player characters in a massively multiplayer online game. The upshot of these discussions was that by emulating nature, developers can automatically get NPCs that look and act more naturally. The standing-room-only crowd at this presentation was a testament to developer interest in learning to build AI that acts more like an actual organic system might.

Learning from the Military: Team AI

Perhaps more significant than developers picking up tips from Mother Nature is the trend among AI engineers to focus on lessons from the military. In roundtables and presentations, developers expressed one common theme they'd found useful in solving their AI design problems: structuring their AIs into layers or groups to better reflect operational doctrine and its abstraction of basic battlefield problems. These layers go by a variety of names depending on the application but can be loosely defined as strategic, operational, and tactical. While not every game necessarily needs or uses all three levels (STARFLEET COMMAND, for example, uses two levels, the Admiral and the Captain), this is the conceptual model that developers say they are using as a starting point.

The strategic level of a layered AI is the one that sets the goals. In a sports game, this is the manager who decides, "Take the end field." In a real-time strategy game, this is the Emperor who decides to attack player X within 10 turns.

The operational layer of a layered AI builds the plans for achieving those goals: how many troops are needed to take a given city, when transports need to be available and where, and so on. The lofty, generalized goals of the strategic layer are

ABOVE. The trend of squad-based games such as STARSIEGE has forced developers to find ways to make groups of individuals work together. LEFT. REPUBLIC features "level-of-detail AI," which takes an algorithmic approach to keeping runaway AI processing in check.

broken down into smaller, testable, achievable subgoals, and resources are allocated to make them happen.

The tactical AI is pretty much what it sounds like, the basic AI at the unit level of the game. It could be the AI controlling an individual football player who decides to snap left instead of driving straight. It could be the AI of an NPC in a first-person shooter who decides to set up an ambush to wait for the player. While responding to the direction given to it by the operational AI, this is also the level at which there are individual motivations, needs, and decision making which often make or break the suspension of disbelief that is so very important to the game-playing experience. If a unit does something dumb here, this is where the player will notice it.

Much of this focus on using military tactics is due in part to the wave of squad-level games that have come out in the last couple of years. From S.W.A.T. 3 to STARSIEGE and ROGUE SPEAR to SHADOW WATCH, a number of games focused on squads rather than supersoldiers and forced developers to take a higher-level design look at making groups of individuals work together realistically.

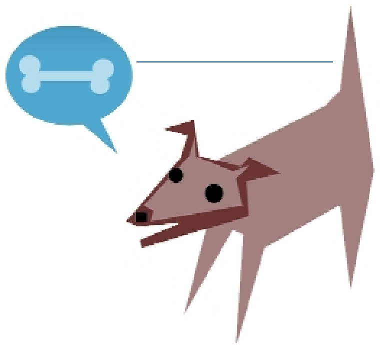
For example, the SHADOW WATCH team gave an impressive presentation on the design and implementation of their AI, and it was apparent that some extremely detailed analysis took place while they were building the AI for this squad-level game. While the strategic AI within the game sets general goals (for example, "assault that building"), the operational or squad-level AI next analyzes the terrain around the building for ideal approaches, maximum cover, and so on. The individual soldiers operating under the tactical AI approach the building in their own unique ways, some crawling, some running, and others wary of the fact that they are low on ammunition. While the game primarily uses a series of prescripted attack styles from which the squads choose, this doesn't take away from the detail of the design. It was very simple for the SHADOW WATCH team to add new attack and movement styles as necessary throughout the game's development, having built the basic engine. These basic action scripts can also be made available to the player for modification and extension, a trend which is becoming ever more popular.

It's interesting to note that while developers are generally very enthusiastic about the possibilities of good team AI, there are problems to overcome. One topic of discussion at the GDC roundtables was the fundamental issue of how to handle what the player doesn't see. There was much unhappiness that players often don't see how clever the team AIs in a given game are. A player doesn't always know, for example, when a teammate AI takes out an enemy soldier that has nearly ambushed the player's avatar. This in turn leads to a difficult design decision, usually late in the development process, that if the player can't see something and it just appears to be random, then maybe it ought to be random in order to conserve CPU resources for the things the player can appreciate. Feedback of some sort between the NPCs and the player is vital.

Trends in Tools and Analysis

Nothing points to the focus on engineering and design going into today's game AI as strongly as the development of tools and analysis within the industry. In my article last year ("Game AI: The State of the Industry," August 2000), I noted that as developers were getting more comfortable with basic pathfinding in 2D and 3D worlds, they were turning to more sophisticated tools such as terrain analysis to make the pathfinding problem more intelligent. This year was no exception, with more games than ever before building on some of the terrain analysis techniques presented at previous GDCs or in the pages of *Game Developer*, such as Dave Pottinger's excellent article, "Implementing Coordinated Movement" (February 1999).

Briefly, terrain analysis is a technique in which the AI analyzes the game map to look for various natural features such as choke points, ambush locations, and so



on. While this comes naturally to most humans, it's not at all obvious to the AI. Good terrain analysis can provide the AI with several resolutions of information about the game map that are well suited to the various AI levels of decision making. The problem is complicated by the extreme popularity of randomized and user-built maps. It's often a toss-up whether the analysis should be done when a level is loaded, which slows down gameplay, or during run time, which means the AI is always working with an incomplete picture.

This focus popped up in a number of presentations and games. The Ensemble team themselves completely revamped their terrain analysis techniques while working on AGE OF EMPIRES II: THE CONQUERORS, resulting in AI that makes better use of terrain, choke points, and other level features than ever before. The SHADOW WATCH team cited Pottinger's *Game Developer* article as inspiration for much of their team-based AI battlefield analysis that went into that game. William van der Sterren made an impressive presentation that built upon many of those same techniques within a QUAKE 2 environment to build a sniper-based game that is uncanny in its ability to select the perfect place for an ambush.

Level-of-detail AI (LODAI). One of the more interesting techniques that generated widespread buzz at the conference was Demis Hassabis's presentation of what he called "level-of-detail AI" (LODAI) that is used in Elixir's upcoming game, REPUBLIC. A technique he developed to control runaway AI processing in the game, LODAI focuses on an algorithmic approach to expanding and collapsing AI processing that allows the bulk of the CPU resources to be allocated to the areas directly in view and control of the player. Off-screen AIs such as those being used in other cities don't just run less frequently than the AIs immediately before the player, as is common in many FPS and RTS games. Rather, LODAI techniques ensure that those off-screen AIs are abstracted using algorithms developed by Hassabis's team that can provide the same general and believable results as the more detailed algorithms, but without the processing overhead. Important details, such as dress colors for important individuals within the game, are maintained over and above the higher-level AI algorithms using a technique called augmentative transitive networks, a somewhat proprietary approach that Hassabis himself designed.

It was interesting to see how quickly the topic of LODAI was picked up at the roundtables. Most developers felt that LODAI was more of a tuning issue than anything else, though there was general agreement that an algorithmic approach, if workable, could be a boon to saving processing cycles. It turns out that the problem is one faced by a number of developers, as a variety of approaches beyond those Hassabis outlined in his talk were discussed by the attendees on how they are handling LODAI in their projects. Current solutions range from upping the priority of individual AI threads to building the AI in an event-based fashion so that agents far from the player don't get as much processing time. Nobody had yet explored the idea of an algorithmic approach (or at least none said so publicly). Most developers felt that LODAI was ultimately something that they won't worry about much during development proper, but that is important to the overall design.

Toolkits and SDKs. As mentioned in previous years, developers have a great deal of interest in AI SDKs and toolkits, but few are actually taking the time to try any out for themselves. When the subject was broached at the roundtables, nearly every attendee wondered if anybody else had used or was considering using one. It turned out that nobody was using any kind of AI SDK or middleware for pretty much the same reasons as in previous years — they didn't feel they were practical, or currently offered sufficient benefits. We discussed the various AI SDKs currently available on the market (Mathematiques Appliquees' DirectIA, LouderThanABomb's Spark!, and the new Biographic Technologies' Autonomous Character Engine plug-in for Maya) and their potential benefits, but beyond the basic information, nobody had used or considered using them for any project. It is apparent that if AI middleware is to gain widespread acceptance, the developers of these toolkits still have a lot of convincing to do. It will have to be very clear that using the SDK will save time and money.

There's nothing like the state machine. It's beginning to sound like a worn record from year to year, but once again developers at GDC 2001 described their game AI as not being in the same province as academic technologies such as neural networks and genetic algorithms. Developers continue to use simple rules-based finite- and fuzzy-state machines for nearly all their AI needs. Their basic simplicity makes these approaches far easier to understand, debug, and explain to the level designer, and they work well in combination with the types of encapsulation seen in games using A-Life techniques as mentioned above. Newer tools such as scripting are simply seen as ways that provide developers and level designers with more flexibility in building and tuning their state machines.

Exotic technologies maintain a slim toehold. Does this mean that developers are stuck in a rut? Could it be that while they're designing perfectly solid AIs that have extensibility and flexibility and all kinds of wonderful features, they're completely overlooking the possibilities of more exotic AI technologies? Not really. There are some developers who are looking at other possibilities while making sure they get their job done in a timely fashion. One developer, who asked not to be identified, is hard at work exploring the possibility of using genetic algorithms as part of his upcoming massively multiplayer online game. He's interested in setting up little "genetics labs" on each player's computer while they're online and using those labs to generate bits of story line for the larger online game itself. These would be based in part on the actions of the players and their interactions with others, then uploaded to the main game server when the players log off for the night. There, the main server would splice together bits and pieces of story lines from hundreds of players to dynamically generate new adventures and story lines for everybody else the next day. This is certainly an intriguing idea, if it can be pulled off.

However, that anecdote is the exception rather than the rule. Exotic AI technologies simply aren't getting much attention anymore. A few developers at the roundtables talked about the possibilities, but they didn't get a great deal of encouragement from the "old hands" who had already explored these technologies. There is general agreement that the fuzzier AI technologies sound really great, but they simply can't deliver the kind of predictable, understandable, and, most importantly, tunable control over a game's AI that developers really need.

Where Game AI Is Going

A favorite topic of discussion among developers at the AI roundtables is where game AI and game AI technology are headed. There are always a lot of opinions on this subject, which of course is what makes it interesting.

Everybody agreed that developers would continue to gain in terms of resources, barring some catastrophic development in physics or speech synthesis that would suddenly pull resources away. There is no doubt that game AI will continue to play an ever more important role in most games, if only because there will be fewer other features differentiating between most products. Most AI developers feel that the industry will continue to move away from hard-coded, monolithic AIs toward designs which are more flexible and better thought out, as witnessed by the talks at this year's GDC. It is widely believed that extensible and "roll your own" AIs will continue to receive some attention so long as players demand it and there is any business case whatsoever, and most feel that we will see these advancements slowly make their way into the multiplayer online games as a matter of course.

Many developers feel that the future of games lies heavily in the online multiplayer market, but this by no means is the end of the need for good AI. Players are going to demand sophisticated NPCs in their online games, and developers will continue to scramble to provide new and better ways for those NPCs to interact in a believable fashion with the player. A delicate balance is necessary here: an NPC that is too smart detracts from the player's game, while an NPC that can't be trusted to keep the horses safe will make players feel like their money is being wasted by the online experience. It is evident that much research is going to be done in this area in the coming years.

One topic which popped up briefly, because the game had only just hit the market when GDC was happening, was the AI in Lionhead Studios' BLACK & WHITE and what that meant for future games. (See Richard Evans's article in this issue, "The Future of AI in Games: A Personal View," based largely on his experiences developing the creature AI for BLACK & WHITE.) The avatars in BLACK & WHITE are somewhat different from those in most games, in that they are heavily influenced by and learn from the actions of the player and the world around them. While they start out fairly ignorant, they rapidly learn both from player training (or beating) and observation of where food is, what kinds of things are acceptable to do and what aren't, and so on. The overall design appears quite sophisticated and powerful. All of this tied in heavily with the perceived need for better NPCs in the online multiplayer games, since these are exactly the characteristics developers are striving for in those games. It was generally expected that there would be much study of the BLACK & WHITE AI in the upcoming year.

Game AI continues to be the single most innovative and fastest-growing segment of the game industry. Developers have a solid feel for what works and are beginning to experiment with designs that let them meet their schedules while still allowing for some experimentation and evolution. There are no more CPU constraints; AI teams are now large enough to allow for all kinds of innovation to come down the road. This is a great time to be a game AI developer.

FOR MORE INFORMATION

BOOKS

The past year has seen several excellent AI references come to the market. Probably the best is the *Game Programming Gems* series by Charles River Media. *Game Programming Gems*, edited by Mark DeLoura, was published in August 2000 and, while not exclusively AI-oriented, has a heavy AI section that has proven very popular. *Game Programming Gems 2* is due out as this issue goes to press and again has a section heavy on AI techniques and tips. Charles River Media has also announced a third volume, tentatively titled *AI Wisdom*, which will use the same "Gems-style" format focusing exclusively on game AI. That book is slated for publication at GDC 2002.

WEB SITES

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

www.gameai.com. The author's own site, dedicated to all things game AI related. Provides links to other AI resources, reviews on AI implementations in games already on the market, and archives of various Usenet threads.

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

www.gdconf.com/archives/proceedings/2001/homepage.htm. The proceedings for all of the many excellent AI design papers presented at GDC 2001 can be found at the conference web site.

www.gamedev.net. Another excellent site dedicated to all aspects of game development, it has an extensive list of resources and an active discussion group on game AI.

www.pcai.com/pcai. *PC AI* magazine has a marvelous web site crammed with all kinds of useful AI resources. From sample apps to research papers, you can find it here.

Return

The Future of AI in Games: A Personal View

In this article, I will describe some AI techniques which, I believe, will become increasingly commonplace in computer games. The conclusions I draw are based on my experience developing the minds of the creatures in Lionhead Studios' Black & White, a god game with lofty ambitions released earlier this year to critical and commercial acclaim.

We'll start by looking at the creatures in Black & White, and then go on to speculate about what sorts of interesting agents we can expect in computer games in the next few years.

The creatures in Black & White had to fulfill two very different requirements. First, we wanted users to feel they were dealing with a person. The creatures had to be plausible, malleable, and lovable. Second, the creature had to be useful to players in their many quests and goals. In Black & White the creatures aren't just toys you experiment with, they can be trained to be invaluable helpers in the campaign.

To my knowledge, this combination of features has not been attempted before. There are titles such as Creatures and The Sims in which you feel you are dealing with passably plausible agents, but these packages, excellent as they are, are more like sandboxes than games: they are pure goal-less simulations in which the entertainment is to be gained from experimentation, not from progressing through a series of quests. And then there are games such as Daikatana, in which the player's character is given helpers to aid on the quest, but in these games the helpers are just state machines, hard-coded for the particular task at hand.

At first glance, there seems to be some conflict between Black & White's two requirements. The person-like requirement implies the creatures are autonomous, whereas the usefulness requirement seems to preclude too much autonomy. Later on we shall see how this conflict was "resolved." To start, let's look at the first requirement, making persons out of creatures.

Making a Person: The Architecture of an Agent

In order for the player to see his creature as a person, the creatures had to be psychologically plausible, malleable, and loveable.

Psychologically plausible agents. To make agents who were psychologically plausible, we took the belief/desire/intention architecture of an agent, fast becoming orthodoxy in the agent programming community, and developed it in a variety of ways. The underlying methodology was to avoid imposing a uniform structure on the representations used in the architecture, but instead to use a variety of different types of representation so that we could pick the most suitable representation for each of the very different tasks (see Marvin Minsky's paper on Causal Diversity in For More Information). Beliefs about individual objects were represented symbolically as a list of attribute-value pairs, beliefs about types of objects were represented as decision trees, and desires were represented as perceptrons. There is something attractive about this division of representations — beliefs are symbolic structures, whereas desires are more fuzzy.

The augmented belief/desire/intention architecture used by creatures in Black & White is shown in Figure 1. Creatures use desires, beliefs, and opinions (which are like universally quantified beliefs) to construct an overall plan. For instance, the creature might decide to attack a certain town. Next, he refines his plan from having a general goal to using a specific action, such as deciding to fireball a particular house in that town. Finally, he breaks that action into a number of simple subtasks which are sequentially executed.

To make a plausible agent, there must be an explanation of why he is in that particular mental state. In particular, if an agent has a belief about an object, that belief must be grounded in his perception of that object. Creatures in Black & White do not cheat about their beliefs — their beliefs are gathered from their perceptions, and there is no way a creature can have free access to information that he has not gathered from his senses. I call this requirement epistemic verisimilitude.

Further, if a creature wants something, there must be an explanation of why he wants it. For example, if the creature is angry, it might be because he has been watching you being destructive and has

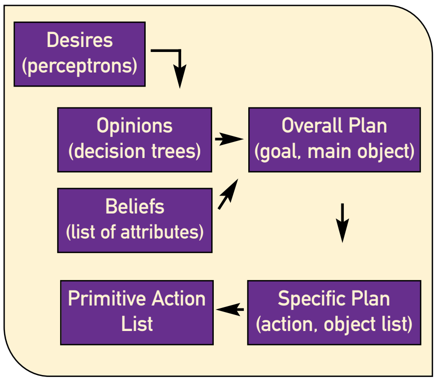

Desires (perceptrons)
Opinions (decision trees)
Overall Plan (goal, main object)
Beliefs (list of attributes)
Primitive Action List
Specific Plan (action, object list)

FIGURE 1. The belief/desire/intention architecture used for creatures in Black & White.

RICHARD EVANS | *Richard is head of artificial intelligence at Lionhead Studios. He studied philosophy at King's College, Cambridge, and then went on to do an M.Sc. in AI at Edinburgh. At the IBM Research Centre, Richard worked on computational natural-language processing. He has also worked as a research consultant at the Institute of Economics and Statistics, Oxford University. Before he joined Lionhead, Richard tinkered with AI techniques in his bedroom.*

decided to copy you; or, the creature might grow angry after getting hurt. Each desire has a number of different desire-sources which jointly contribute to the current intensity of the desire. For example, there are three possible explanations of why a creature could be hungry: his energy could be low, he could have seen something that he knows is tasty, or he could be sad. By tweaking the thresholds of these three sources, you can make a variety of different sorts of personalities: creatures who only eat when they are starving, creatures who are greedy, even creatures who binge-eat when they are depressed (see Figure 2).

Malleable agents. We wanted the creatures to be malleable in many different ways; we wanted them to learn many different types of things, and we wanted there to be many different types of situations that would prompt learning. "Learning" covers a variety of very different skills:

- Learning that (for example, learning that there is a town nearby with plenty of food)
- Learning how (for example, learning how to throw things, improving your skill over time)
- Learning how sensitive to be to different desires (for example, learning how low your energy must be before you should start to feel hungry)
- Learning which types of object you should be nice to, which types of object you should eat, and so on (for example, learning to only be nice to big creatures who know spells)
- Learning which methods to apply in which situations (for example, if you want to attack somebody, should you use magic or a more straightforward approach?)

The architecture was designed to allow all these different types of learning. Learning can be initiated in a number of very different ways:

- From player feedback, such as stroking or slapping the creature.
- From being given a command. When the creature is told to attack a town, the creature learns which sort of towns should be attacked.
- From the creature observing others, such as the player, other creatures, or villagers.
- From the creature reflecting on his experience: after performing an action to satisfy a motive, seeing how well that motive was satisfied, and adjusting the weights representing how sensible it is to use that action in that sort of situation.

The architecture was designed to allow all

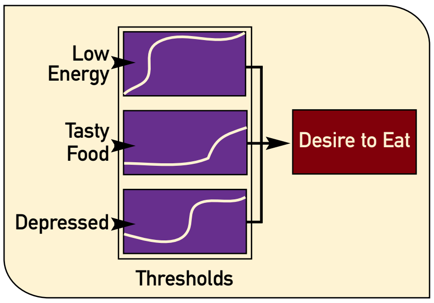


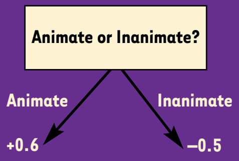
FIGURE 2. Desire sources contribute to the intensity of desire experienced by a creature.

Learning Opinions: Dynamically Building Decision Trees

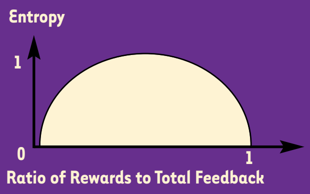
How does a creature learn what sorts of objects are good to eat? He looks back at his experience of eating different types of things and the feedback he received in each case, as well as how nice they tasted, and tries to "make sense" of all that data by building a decision tree. Suppose the creature has had the following experiences:

What he ate	Feedback — "How nice it tasted"
A big rock	–1.0
A medium rock	–0.5
A small rock	–0.4
A tree	–0.2
A cow	+0.6

He may build the following simple tree to explain this data:



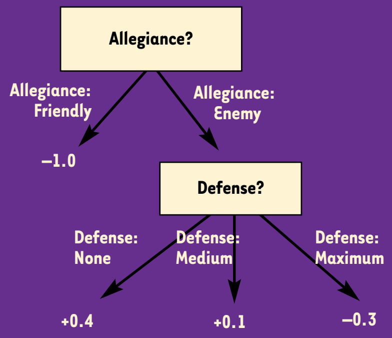
A decision tree is built by looking at the attributes which best divide the learning episodes into groups with similar feedback values. The best decision tree is the one which minimizes entropy. The entropy is a measure of how random the feedbacks are. If the feedbacks are always 0, there is no randomness; entropy is 0. If the feedbacks are always 1, again there is no randomness and entropy is 0. But if the feedbacks alternate between 0 and 1, the feedback is random and unpredictable; entropy is high. We build a decision tree by choosing attributes that minimize the entropy in the feedback.



To take a simplified example, if a creature was given the following feedback after attacking enemy towns:

What he attacked	Feedback from player
Friendly town, weak defense, tribe Celtic	–1.0
Enemy town, weak defense, tribe Celtic	+0.4
Friendly town, strong defense, tribe Norse	–1.0
Enemy town, strong defense, tribe Norse	–0.2
Friendly town, medium defense, tribe Greek	–1.0
Enemy town, medium defense, tribe Greek	+0.2
Enemy town, strong defense, tribe Greek	–0.4
Enemy town, medium defense, tribe Aztec	0.0
Friendly town, weak defense, tribe Aztec	–1.0

Then the creature would build a decision tree for Anger like this:



The algorithm used to dynamically construct decision trees to minimize entropy is based on Quinlan's ID3 system (see For More Information).

these different ways in which learning can be initiated.

All these different types of learning, and different types of occasions which prompt learning, coexist in one happy bundle. I will only go into detail about one of these types of learning, learning which types of objects are most suitable for various different desires. (See the sidebar "Learning Opinions: Dynamically Building Decision Trees.")

Lovable agents. We wanted the player to feel emotionally attached to the creature. We soon realized that empathetic attachment is intrinsically reciprocal. The reason why it is childish to feel emotionally attached to your teddy bear is because your teddy is not going to reciprocate. Conclusion: If you want the player to get attached to the creature, you must first ensure the creature is empathetically attached to the player.

Agents in computer games are at best like severely autistic people. They are capable of perceiving and predicting the behavior of objects in the world, but incapable of seeing other people as people — incapable of building a model of another agent's mind which could be used, to great effect, to predict its actions.

In BLACK & WHITE, the creature's mind includes a simplified model of the player's mind. He watches what actions the player is performing and tries to make sense of those actions by ascribing goals to the player which would explain those actions. He stores a simple personality model of the player, which he uses in decision making. He also has goals which relate directly to his master: the desire to help his master, the desire to play with his master, and the desire for attention.

Making a Useful Person: Autonomy Can Go Too Far!

The creatures in BLACK & WHITE had to be person-like, but they also had to be useful. The person-like requirement implies the creatures are autonomous, whereas the usefulness requirement seems to preclude too much autonomy. How can we resolve these conflicting requirements? The solution we arrived at was that creatures start off completely autonomous, but over time, through training, the player can mold them so that they only do what the player wants them to do. This gives players an enormous feeling of satisfaction that they have trained the creature to actually be useful in the game. The downside is that your creature loses something of his charm the more you train him. He becomes more focused on a few goals in a few situations on a few types of objects. As he becomes more useful, he becomes more "robotic."

A related issue, equally difficult to balance, is to what extent should the creature start off being a blank slate, or should we give him knowledge at birth to enable him to be more useful? When the first creature was put into the game world, he just stood there, staring at his feet. I was expecting him to exhibit more interesting behavior than this and was rather disappointed. After debugging, it turned out the reason he was just standing there staring at his feet was that he was very hungry, had looked around for something to eat, and had chosen himself as the nearest candidate foodstuff. He kept trying to pick himself up, failing, and trying again. Now, we could have just left this in there and had creatures learn over time that they cannot eat themselves, and learning from failure. However, this would have been learning for learning's sake, not interesting in a game world, so we gave the creature the additional knowledge that he cannot eat himself.

Creatures can also learn the healing spell, and early on in development we had creatures learning which sorts of things to try to heal. They would try anything — they would even try to heal rocks. Again, this sort of thing is flexibility for flexibility's sake, and just means that the initial creatures are more dysfunctional and the players have more to do to train them. In the end, we just insisted that the only sort of thing you can heal are living things and houses — there is still enormous room for variety of behavior within these more sensible constraints.

Future Directions: Extrapolating from BLACK & WHITE

Now we shall move on and consider how the concept of a semi-autonomous helpful agent can be expanded. These are the things we are hoping to do next:

Person-like agents. What could we do to make more realistic agents? First, we could give them an infinite number of goals. Creatures in BLACK & WHITE have a finite number of goals. People, by contrast, have an indefinite number of goals: you can want to write a story, you can want to write a detective story, you can want to write a detective story in which the detective is a woman, you can want to write a detective story in which the detective is an old, cantankerous woman, and so on. Desires have the same combinatorial structure as human language itself, and this needs to be captured explicitly.

Additionally, there needs to be an infinite number of ways of satisfying goals with real-time planning. Creatures in BLACK & WHITE use plan libraries to work out what to do. The creatures know (at birth, if you like) that there are *k* ways of satisfying a particular goal. When they try to satisfy that goal, they just go through those *k* ways, seeing which is most suitable in the current situation. People, by contrast, sometimes use real-time planning, where they generate novel ways of satisfying a goal by trying out various options and considering the results of those actions. This is computationally expensive but gives the agent a freedom and flexibility currently missing.

What advantages would these additions give us? They would give us an indefinite amount of flexibility in possible behavior. Even with a finite number of different animations the agents can play, these additions would enable an indefinite number of different ways of sequencing these animations. (Important parallel: There are only a finite number of words in a language, but an infinite number of possible sentences.)

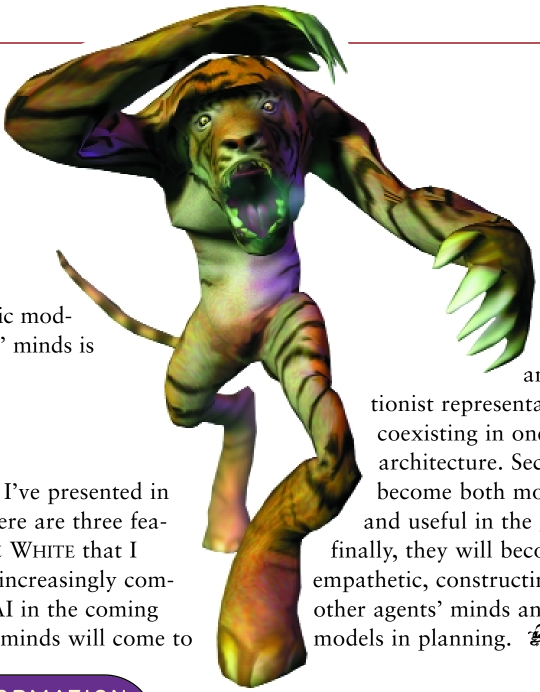
Empathetic agents. If we want to make more plausible agents, we enrich the mental model of the agent. If we want to make more empathetic agents, we enrich the mental model that the agent uses to model other agents. (These two are quite distinct: the latter is invariably going to be simpler than the former, for space-efficiency reasons. The agent is going to have models about lots of different agents, so these models should be small.) The creatures in BLACK & WHITE have simple models of other agents' minds; they just model the desire part of the architecture. Wouldn't it be nice to add more?

The trouble is that the more we enrich the agent's model of other agents, the harder it is for the agent to figure out what the other agent is thinking. For instance, suppose our agent's model of another agent includes data about the other agent's beliefs as well as his desires. Then we have made the task of understanding the other agent considerably harder, because there will be more models which fit the data, and it will be harder to figure out which is best.

Suppose, for instance, that an agent fails to eat the apple. This might be because he hasn't seen the apple (and consequently has no belief about it), or because he doesn't like apples, or because he just isn't hungry. Which of these is the right explanation? We can't tell until we have seen a lot of examples. (This problem just doesn't arise if you keep an excessively simple model of other agents: if you just model them as a bunch of desires, then the only possible explanation is that he isn't hungry.)

There are proposed solutions to this in philosophical literature: the principle of charity solves the apple problem by assuming the agent's beliefs are correct, but if we are going to assume this across the board, then there is no point in modeling beliefs at all. We can sidestep this problem by keeping a number of alternative possible models of an agent's mind, but this requires even more storage space. These problems are not intractable, but do show that making a deeper model of empathy isn't particularly trivial.

What advantage would a deeper model of empathy give us? If agents had a richer model of other agents' minds, they could use it to understand language. As the philosopher Paul Grice argued, understanding that agent X is saying P involves understanding that X wants me to believe P, and understanding that X wants me to believe that X wants me to believe P (and so on). More generally, interesting social behavior involves the modeling of other agents' minds. (For example, deception involves understanding that if you say P, X will believe P, and will then do A.) At the moment, communication with computer agents in role-playing games involves choosing from a small, finite list of canned sentences. Empathetic modeling of other agents' minds is the key here.



Summary

Of the concepts I've presented in this article, there are three features from Black & White that I expect will become increasingly commonplace in game AI in the coming years. First, agents' minds will come to include both symbolic and connectionist representations, happily coexisting in one unified architecture. Second, they will become both more person-like and useful in the game. And finally, they will become more empathetic, constructing models of other agents' minds and using these models in planning.

FOR MORE INFORMATION

BELIEF/DESIRE/INTENTION ARCHITECTURES

Rao, Anand S., and Michael P. Georgeff. "Modeling Rational Agents within a BDI-Architecture." *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, 1991.

http://citeseer.nj.nec.com/rao91modeling.html

MARVIN MINSKY ON CAUSAL DIVERSITY

www.ai.mit.edu/people/minsky/papers/CausalDiversity.html

QUINLAN'S ID3: DYNAMIC LEARNING OF DECISION TREES

Quinlan, J.R. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

http://yoda.cis.temple.edu:8080/UGAIWWW/lectures/C45

PAUL GRICE'S THEORY OF MEANING

Grice, H. P. "Meaning." *The Philosophical Review.* Vol.66 (1957): 377–388.

Return

FIGURE 1. Good old terra firma, minus the firma.

Collision Detection Using Ray Casting

Game physics can be divided up into three distinct and (usually) separate phases: simulation, collision detection, and collision response, colloquially known as which way do I go, what did I hit, and where do I go from here.

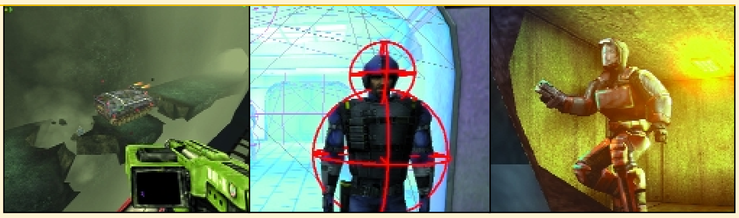
Collision detection is generally the bottleneck of the three. It is a math-intensive task to determine when something hits something else, and even today's high-horsepower PCs and consoles can choke on the load. To throw gas on that fire, games are continually getting more complicated, with more objects to simulate, more polygons to collide against, and more sophisticated physics. As the physics programmer on Volition's recently released Playstation 2 FPS, RED FACTION (RF), I can tell you that flame burns hot.

RF takes physics complexity to a new level, beyond most current games. Volition's proprietary GeoMod engine is designed to allow real-time arbitrary geometry modification, something that hasn't been done in a 3D game before. This means that a player can point a rocket at any wall and create a gaping hole to jump through. The big problem this poses for collision detection is that it eliminates a lot of the preprocessing tricks that can be used to optimize calculations. A BSP tree, for example, is not something that you can recompute between frames if the level geometry changes.

A second design challenge for RF is a very ambitious physics engine. When the player blows up a bridge (for example), big chunks of rock should break off and tumble to the ground in a realistic way, maintaining angular momentum and picking up spin from friction with the ground. These computations take up a lot of CPU cycles and require a pretty accurate collision detection system to look right.

So, we can use no preprocessing and we need high accuracy. What kind of collision detection is going to work with these terrifying caveats? Let's take a brief look at some of the options.

TIM SCHROEDER | *Tim is a programmer working at Volition in Champaign, Ill., and a card-carrying member of the Canadian brain drain. When the big Canuck isn't beating his head against a wall trying to solve collision detection problems, he can usually be found doing something sporty. He can be reached at tims@volition-inc.com.*



Intersection Testing

By far the most popular form of collision detection in games, intersection testing is conceptually simple and robust. It works by checking an object's desired position and determining whether it will intersect the world or another object. If it will intersect, the object is backed up and tested again, recursing until no collision occurs. The common algorithm for this process is a binary subdivision of the distance between the last valid position and the desired position of the object. The beauty of binary subdivision is that it allows you to find a happy medium between accuracy and speed by tuning the number of iterations.

There are a number of methods for doing the actual intersection test, from axis-aligned bounding boxes to convex hulls employing Voronoi regions. There are also a number of preprocessing steps that can make the intersection testing faster. A common method is to generate a BSP tree of the world geometry. This simplifies the intersection test to a series of plane equations and nicely subdivides the world into a hierarchical tree of valid areas. You can also use a simplified set of planes to define the boundaries of the world separate from the rendered geometry to cut down on the complexity of the collision detection.

The main problem with intersection testing is that for any moving object you can choose a sufficiently large frame time and a sufficiently high velocity such that the object moves completely from one side of an obstacle to the other without intersecting it. Thin walls pose a significant problem for that reason. This was a big strike against that method when evaluating schemes for RF, since the player can make all manner of malformed geometry by blowing away chunks of the world. In a controlled environment restrictions can be put on level designers to solve this problem, but it's impossible to do the same for the player.

Ray Casting

Ray casting is fundamentally different from intersection testing in that it projects the object along the path from its last valid position to its desired position and attempts to determine the exact time that the object collides with an obstruction (see Figure 2). This is a fairly simple operation with a particle, but it becomes much more expensive with objects of volume. To cut down on complexity, the object is generally approximated with simple shapes, such as a bounding box, a collection of spheres, or possibly a convex hull.

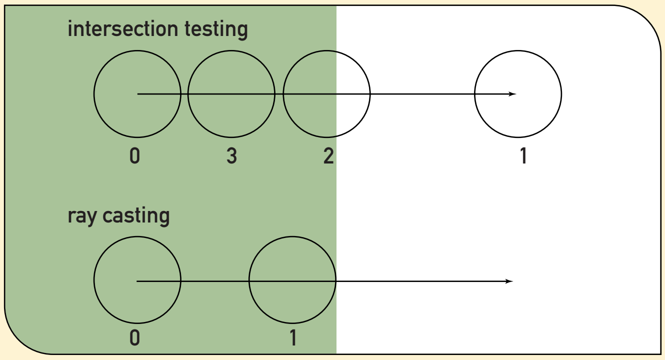


FIGURE 2. Intersection testing versus ray casting.

The advantage of this system is that it only requires one iteration and the results are more accurate than multiple iterations of binary subdivision intersection. The drawback is that the collision calculation is more expensive than an intersection test for objects of equal complexity. The trade-off between speed and accuracy becomes more apparent at high velocities.

The preprocessing steps that are applicable to intersection testing are just as applicable to ray casting. Unfortunately, most of them assume a static world and thus are not applicable to RF.

Our Design

After evaluating the options for RF, ray casting came out the clear winner. Even so, there was a whole host of optimizations we needed in order to make the collision detection speedy enough. In the rest of this article, I'll detail the collision detection methods implemented in RF. But first, let's look at a roadmap of the system as a whole.

At the highest level, the world in RF is divided into a series of rooms separated by portals. This is a common rendering optimization that nicely breaks the world up into bite-sized chunks. The polygons located in a room are organized into a binary tree of bounding boxes. At the lowest level, the polygons themselves each have individual bounding boxes around them. In order to collide an object with a polygon, the object's bounding sphere is used. The object first has to pass through the right room, then the right section of the bounding box tree, and finally the bounding box of the polygon.

Now that you have an idea of how our collision detection system works, let's start with the basics and work our way up.

Simple Particle/Polygon Collisions

The simplest form of ray casting boils down to a particle (which has no volume) moving through space, and a polygon. There are two tests to perform when doing collision detection with a particle:

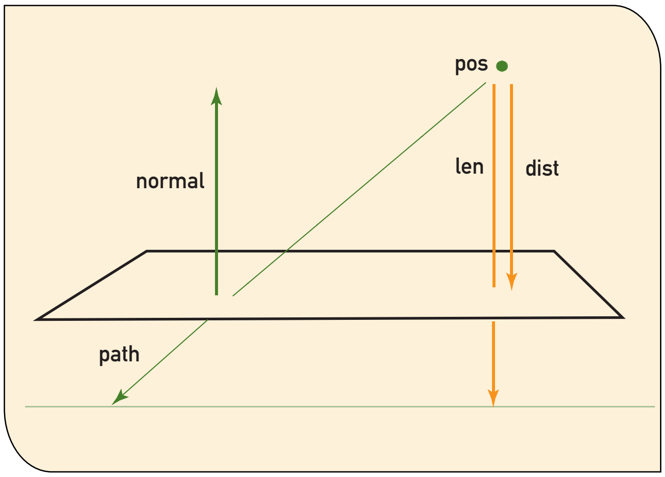
1. Where does the particle cross the plane of the polygon?
2. Is that collision point inside the polygon?

The first test is greatly simplified if the plane equation of the polygon is known. In RF each world face stores its plane equation to keep from having to recompute it every time it is needed.

Let me clarify what constitutes a collision. A collision occurs when an object passes from the front of a polygon to the back. Objects moving in the other direction are not considered.

Now that we're clear, let's collide our particle with the plane. To do this we need the initial position of the particle, its path during this frame, and the plane equation of the polygon. The algorithm is very simple:

1. Find the distance (`dist`) from the particle to the plane.
2. Find the length (`len`) of the path vector along the normal of the plane.
3. Determine the time of the collision using distance divided by length.



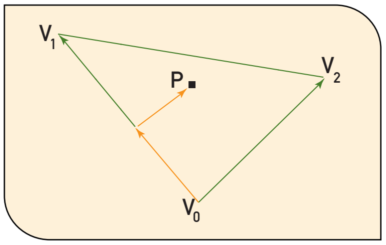


FIGURE 3 (above). Colliding with the plane. FIGURE 4 (right). Determining if the hit point is inside the polygon.

If the time of the collision is negative, the plane is behind the particle. If the time is greater than 1, the plane is too far away to hit this frame. The code to find the collision point follows. Note that `path` is the vector from the current valid position of the particle to the desired position, and it is not normalized.

```
bool collide_line_plane(vector &pos, vector &path, plane &pl,
                        float &time)
{
   float dist, len;

   dist = pl.distance_to_point(pos);
   if( dist < 0 ) {
      // behind plane
      return false;
   }

   len = -(path * pl.normal); // Vector dot product
   if( len < dist ) {
      // moving away from plane or point too far away
      return false;
   }

   time = dist/len;
   return true;
}
```

If the particle has crossed the plane this frame, the time variable will contain the fraction of the frame time at which the collision occurred. The collision point P can then be found with the following parametric equation:

$$P = (\texttt{path} \times \texttt{time}) + \texttt{pos}$$

Now we need to see whether that hit point is within the boundaries of the polygon. To do that, we reduce the problem to two dimensions by eliminating one of the world axes from our calculations. We eliminate the dominant component of the normal to get the best projection onto a plane formed by two of the three axes. Representing the vectors as arrays, those two remaining axes are identified by the indices j and k. We then compute 2D vectors for the hit point P and two vertices V_1 and V_2 relative to the third vertex, V_0:

$$A = V_1 - V_0$$
$$B = V_2 - V_0$$
$$C = P - V_0$$

This allows us to describe the hit point as the sum of two of the edges of the polygon:

$$C = \alpha A + \beta B$$

We can now solve for α and β using determinants:

$$\alpha = (B_k C_j - B_j C_k) / (A_j B_k - A_k B_j)$$
$$\beta = (A_j C_k - A_k C_j) / (A_j B_k - A_k B_j)$$

Three conditions must be satisfied for the point to be inside the polygon:

1. $\alpha \geq 0$
2. $\beta \geq 0$
3. $\alpha + \beta \leq 1$

If all three conditions are true, we have found a collision. Note that this algorithm only works on triangles. N-sided polygons will have to be triangulated first or the algorithm modified to loop through the vertices of the polygon three at a time. For further detail on this algorithm, see For More Information.

Crank up the Volume

Colliding rays with polygons is all well and good for a particle system, but our objects have some volume, so we need something more complex. There are a number of possible tech-

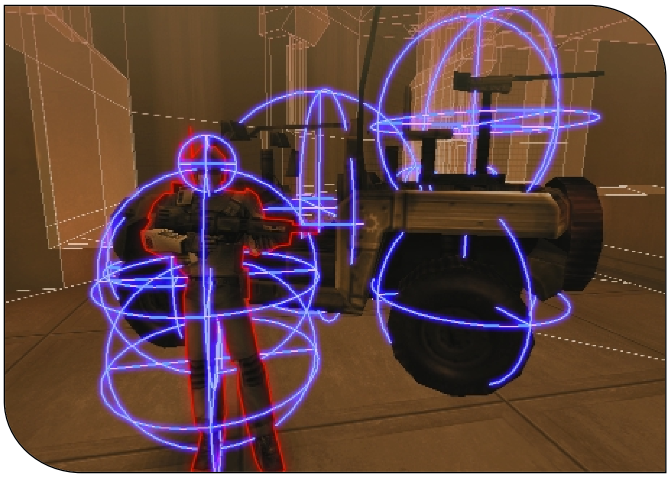
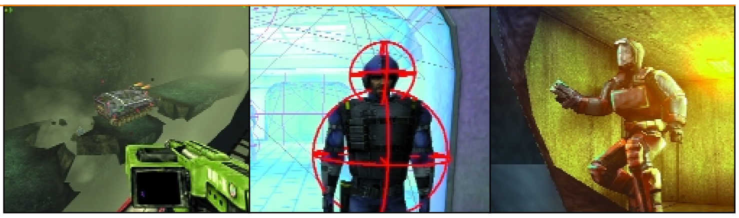


FIGURE 5. Objects are just collections of spheres.



niques for approximating an arbitrary object to use in collision detection. These include bounding boxes, collections of linked particles, and convex hulls. In RF, objects are approximated by a collection of spheres.

Spheres have a number of wonderful properties for collision detection, not the least of which is that a sphere has the same shape from every viewing angle. This makes colliding a sphere with a polygon very similar to colliding a particle. When a sphere bumps into a plane, it's always the point closest to the plane that touches, so if we find that point we can then pass it and the sphere's path into the `collide_line_plane()` function described earlier to determine the hit point. The point on a sphere (centered at `pos`) that's closest to the plane `pl` is found using the equation:

P = `pos` – (`pl.normal` $\times$ r)

Sphere/edge collision. The complication, however, is that if the hit point isn't located within the polygon, it is still possible for the sphere to have hit just an edge of the polygon, or even just a vertex. Let's look at the edge case first — we need a function that can determine the collision of a sphere with a line segment (an edge of the triangle).

Two conditions are true when a collision occurs: First, the distance from the line to the sphere center is equal to the radius of the sphere, and second, the vector from the hit point to the sphere center is perpendicular to the line (in other words, their dot product equals 0). If we put our conditions into equations we get:

1. (`hit_pos` – `hit_point`) • (`hit_pos` – `hit_point`) = r^2
2. (`hit_pos` – `hit_point`) • $(V_1 - V_0) = 0$

Where the position of the sphere at the time of impact is given by the parametric equation:

`hit_pos` = (`path` $\times$ `hit_time`) + `pos`

and the hit point on the edge is likewise given by the parametric equation:

`hit_point` =[$(V_1 - V_0)$ $\times$ `dist`] + V_0

The two things we need to compute are **`hit_time`**, which is the time when the sphere hits the line, and **`dist`**, which is the distance from V_0 along the line to where the sphere touches when they collide. `Hit_time` must be between 0 and 1 for a collision to occur in this frame, and `dist` must be between 0 and 1 for the hit point to be between the end points of the triangle edge. That leaves us with two equations and two unknowns. If you're like me this is when you break out Mathematica.

Once the equations have been simplified and solved, we wind up with a quadratic function in the form $y = ax^2 + bx + c$. The roots of that function are the `hit_time` values for collisions of the sphere with the line subject to our constraints. The coefficients (a, b, and c) of the quadratic turn out to be an unholy concoction of dot products:

$$a = (V_e \bullet V_s)^2 - (V_e \bullet V_e)^2$$
$$b = 2 \times [(V_d \bullet V_e) \times (V_e \bullet V_s) - (V_d \bullet V_s) \times (V_e \bullet V_e)]$$
$$c = (V_d \bullet V_e) \times (V_d \bullet V_s) + [r^2 \times (V_e \bullet V_e)] - [(V_d \bullet V_d) \times (V_e \bullet V_e)]$$

Once we have a, b, and c we can plug them into the quadratic formula and see what comes out. The quadratic formula, which you probably thought would never come in handy outside a high school math class, is:

$$roots = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If $b^2 - 4ac$ (the determinant) is less than 0, the solution is composed of imaginary numbers, which means that the sphere never got close enough to the line to collide. If the determinant equals 0, there is only one root, which means that the sphere grazed the line tangentially. There is typically no collision response necessary for that case, so we can safely ignore it as well.

If we do get two roots, the smaller is the one that interests us because it is the hit time of the collision. The larger represents when the backside of the sphere would have collided with the line segment on its way past.

Sphere/vertex collision. Unfortunately we're still not done. If the sphere didn't hit the line within the vertices, it could still hit one of its end points, so we need to test for collision with them as well. The good news is that since we test every edge of the polygon, we only need to check the first vertex on each edge. Also, some of the work we did to set up for the edge collision is applicable to the vertex collision.

$$a = V_s \bullet V_s$$
$$b = 2 \times (V_d \bullet V_s)$$
$$c = (V_d \bullet V_d) - r^2$$

All that's left is to plug the new a, b, and c into the quadratic

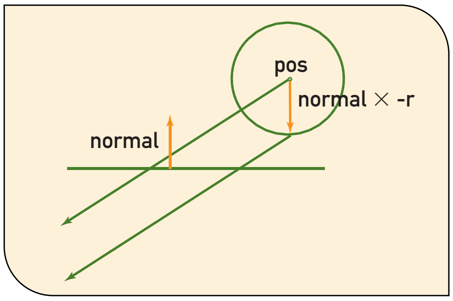
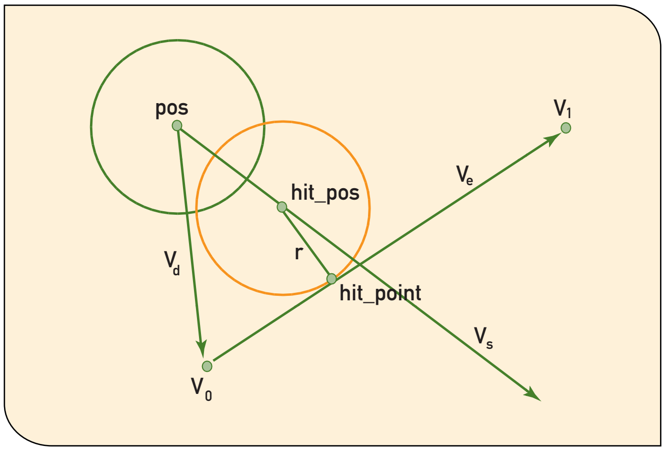


FIGURE 6 (left). Turning a sphere collision into a particle collision. FIGURE 7 (below). Sphere/line segment collision.



formula and find the roots the same way we did for the line.

One thing to note is that sphere/line segment collision tests are by no means cheap, yet they potentially occur three times for every polygon. Further, each line segment in the scene could be tested twice, since polygons share edges. One thing that we did on RF that eased this pain was to check the edge against a bounding box around the sphere's path. The line segment/bounding box test quickly culls out the edges that couldn't possibly collide and makes a nice segue to my next point.

Putting Polygons into Boxes

Now we have the tools for throwing spheres around and colliding them with the world. Unfortunately, unless your world consists of just a few boxes, this isn't going to be fast enough to run in real time. The big problem is that planes are infinite, so if you don't have some notion of locality, you are forced to do expensive collision tests with every polygon in the entire world. The collision detection system needs a fast way to cull the number of faces it has to check. Enter axis-aligned bounding boxes (AABBs).

The simplest collision test you can perform is the intersection of two AABBs, which takes at most six comparisons. If each world face has a bounding box around it, and we create a bounding box around the path of the sphere, then we can quickly and easily tell whether that object can possibly hit the polygon. In fact, it is usually faster to compute these face bounding boxes on the fly and use them for culling than it is to do the expensive sphere/face tests.

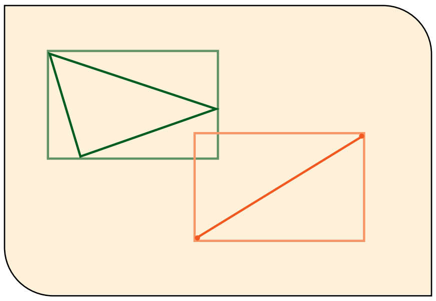
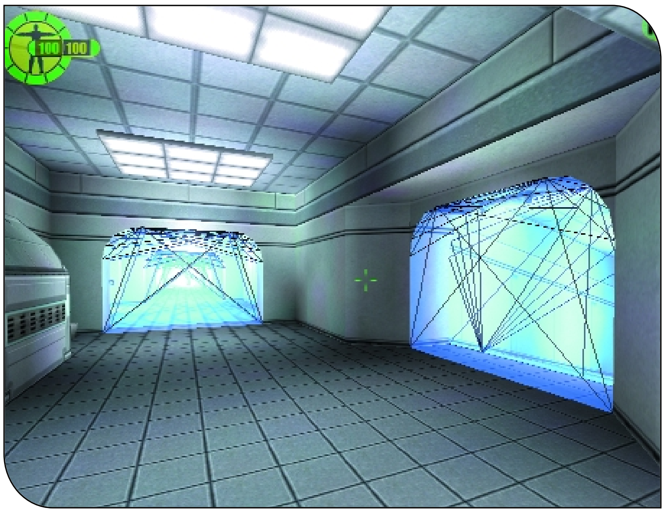


FIGURE 8 (right). Bounding box inaccuracy.
FIGURE 9 (below). Rooms separated by portals.



A problem with the bounding box test is that if the sphere moves quickly its bounding box will be quite large, particularly if it moves diagonally to the world axes. A slightly more expensive but potentially much more accurate test is to intersect the sphere's path vector directly with the face bounding box. But when colliding the path of a sphere with the bounding box, the radius of the sphere must be added to the bounding box for this to work properly. The line segment/bounding box intersection function we used in RF eliminates twice as many faces as the standard bounding box intersection test and is only slightly slower.

The way this works is similar to Cohen and Sutherland's 2D line-clipping algorithm (see For More Information). Each endpoint of the line segment is given an outcode based on which sector it is in. The trivially true case, where either of the endpoints is inside the box, and the trivially false case, where both endpoints are on the same side of the box, are found quickly with minimal effort. If further work is required, the outcodes isolate which planes to clip the line segment against.

For each plane of the bounding box that clips the line segment we need to check the intercept point on the plane to see whether it actually falls on the box. This is accomplished by determining the distance along the line segment at which it intercepts that plane. If we are clipping to the bounding box's maximum x plane, for example, this equation is:

$$\texttt{dist} = (V_{1x} - V_{0x}) \,/\, (x_{\max} - V_{0x})$$

And the intercept point would be:

$$\texttt{intercept} = [(V_1 - V_0) \times \texttt{dist}] + V_0$$

In our example, the y and z values for the intercept are then checked to see whether they fall within the minimum and maximum values for the bounding box.

Note that any line segment that intersects the box will cross at least two planes, but finding one is sufficient to know that we can't cull this polygon. The time and position of the intersection can be ignored if the bounding box test is used only for culling.

Putting Boxes into Bins

Instead of thousands of expensive sphere/polygon checks, our engine now does thousands of cheap bounding box checks. That's a big win, but it's still a lot of number crunching. We need a way to cull out more than one polygon at a time so that we don't need to look at every polygon in the level for every potential collision. We'll use some spatial partitioning to narrow down the list of potential collision candidates.

A very simple but effective partitioning system is to create a binary tree of AABB "bins" which hold the polygon AABBs. Each bin contains either two smaller bins or a collection of polygons. Creating the tree is easy, and it can be tuned for performance. Once the tree has been created, bounding box testing becomes a recursive procedure of checking whether an object is within successive child boxes. The creation algorithm is:

1. Create a bin that encompasses all polygon bounding boxes.

2. Divide the bin in half across its longest axis.
3. Use the center of each polygon to determine which bin it belongs in.
4. Add the polygon AABB to the proper split bin.
5. Repeat step 2 for both new bins.

The recursion can be truncated as necessary with a limit on the depth of the tree, a target for the number of polygons in a bin, or a maximum bin volume. The ability to tune the algorithm with these parameters was vital for RF, because the tree has to be recalculated whenever the level geometry changes.

One undesirable property of the resultant tree is that there can be a fair amount of bin overlap if the polygons are large. This means that an object can be found in both child bins at the same time, which defeats the purpose of having the tree in the first place. In practice, however, using the tree drastically cuts the number of AABB checks performed, and the method scales up very well with level size and polygon count.

I mentioned earlier that the RF renderer is room/portal based. It makes sense to utilize that natural partitioning to help out the collision detection. Since all polygons in a level are explicitly linked to exactly one room, it is a simple task to create a bounding box tree for each room. This is also a big win when recalculating a tree after a geometry modification occurs, since only the affected room needs to be recalculated.

Floating-Point Fun

One thing left to mention is how to deal with numerical imprecision. If computers could represent real numbers exactly, there would be no problem implementing mathematical formulae in code. In practice, however, computers are finite and imprecise. Colliding a sphere with a plane when the two are almost touching, for example, is not guaranteed to work without error. Because of this imprecision it is a good idea to incorporate some judicious fudging into the equations. It is in the fudge factors that the art of collision detection is found.

The best place I've found to add fudge is to the hit time returned from the collision detection functions. If you collide a sphere with the world and you determine that it hit a wall 60 percent of the way into the frame, you can artificially manipulate that hit time to make your sphere stop short of the wall. You don't want to do that based strictly on time, however, since slow objects would then stop closer to walls than fast ones. A better option is to make this buffer distance-based, and more than that, to maintain the distance from the polygon rather than the hit point.

To adjust the hit time, first find the distance from the current position to the plane of the polygon. Then subtract the buffer distance from that and recalculate the percentage along the original path.

```
len = – (path • normal)
dist = (len × hit_time) – buffer_dist
new_hit_time = dist / len
```

It's important to keep the adjusted hit time from going negative, even though that means that the object is inside the buffer zone. A negative hit time will back the object up and might put it through another polygon behind it, which is much worse.

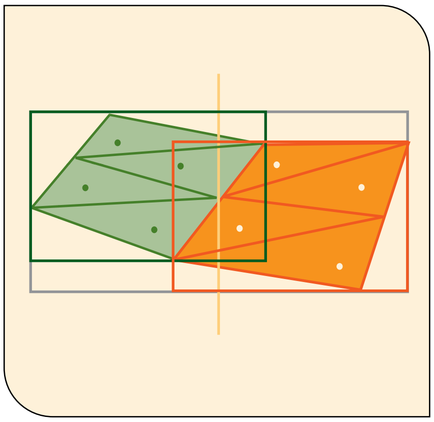


FIGURE 10. Clipping a line segment to an AABB.

Summing Up

Like most things, ray casting has advantages and disadvantages. It works very well within the framework of RED FACTION's unique challenges, but it isn't the solution to every game's collision detection needs. However, even if ray casting doesn't float your particular boat, the optimization techniques used in RED FACTION can be incorporated into other collision detection systems.

Ray casting is a well-researched area of computer graphics, and there are some great resources available on the web. The most useful one I've found is The 3D Object Intersection page (see For More Information), which provides links to various intersection algorithms and source code repositories in a handy grid. Another good resource is David Andsager, but he's generally only available at the Volition offices. The source code for the techniques detailed in this article is available for download from *Game Developer*'s web site at www.gdmag.com.

FOR MORE INFORMATION

RAY-POLYGON INTERSECTION ALGORITHM

Badouel, Didier. "An Efficient Ray-Polygon Intersection." In *Graphics Gems*, edited by Andrew S. Glassner, pp. 390–93. San Diego: Academic Press, 1990.

THE COHEN-SUTHERLAND LINE-CLIPPING ALGORITHM

Foley, James D., and others. *Computer Graphics: Principles and Practice*, 2nd ed. Reading, Mass.: Addison-Wesley, 1996. pp. 113–17.

THE 3D OBJECT INTERSECTION PAGE

www.realtimerendering.com/int

Return

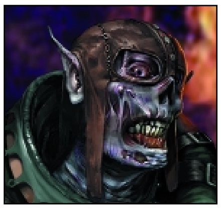
adel chaveleh and denis papp

TimeGate Studios' Kohan

Not everyone can assemble an entire team, make the necessary contacts, and ship a polished game in two years, but we knew it could be done and were determined to do so. TimeGate Studios was founded in late 1998 by executives of large Houston-based corporations in the healthcare, oil and gas, and real estate industries. From day one, TimeGate had access to accounting, human resources, and information technology departments. These are some of the headaches that most startups face, so having access to this structure quickly proved to be a strong point for our new company. This allowed us to concentrate on assembling an extremely dedicated, skilled team.

During those first few months, the game development market was very competitive, and there were several new game development companies being formed. The first obstacle that we had to overcome was being a startup, so we had to set ourselves apart from the rest. With stories of game companies being created and shut down within the course of a year, several people that we spoke to were leery of the label "startup." Their concerns were put to rest after visiting our offices, and seeing how our shareholders were hardcore about gaming, yet experienced in business. After the first four months, the core team was in place and the majority of our project research had been done. It was time to proceed with full-scale production.

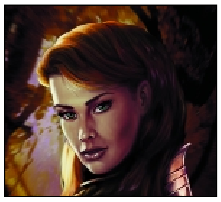
We were huge fans of every genre out there, but we were very partial to the popular real-time strategy (RTS) genre. The one thing that bothered us about RTS games was their emphasis on micro-management that detracted from high-level strategy. Our plan involved combining concepts from multiple genres to create a game that would make people look at RTS games in a whole new light. We wanted a unique game that had mass appeal, yet would also appease hardcore strategy fans. The ambitiousness of these gameplay concepts, combined with the fact that this was TimeGate's first project, presented a serious challenge. However, this challenge was something we were ready to undertake and very excited about.



The story line for Kohan was in development for nearly a decade prior to the start of production. It was the brainchild of

ADEL CHAVELEH | *Adel co-founded TimeGate Studios upon graduating from Louisiana State University in computer and electrical engineering. He made his debut in the computer gaming industry serving as producer on Kohan. When he's not playing Kohan online, you can reach him at adel@timegatestudios.com.*

DENIS PAPP | *Denis is the lead programmer at TimeGate Studios. He obtained his master's degree in computer science from the University of Alberta, where his thesis involved developing one of the first strong poker AIs ("Loki") with the Games Research Group. He started professionally at BioWare Corp. as the lead programmer completing Shattered Steel. He can be reached at denis@timegatestudios.com.*



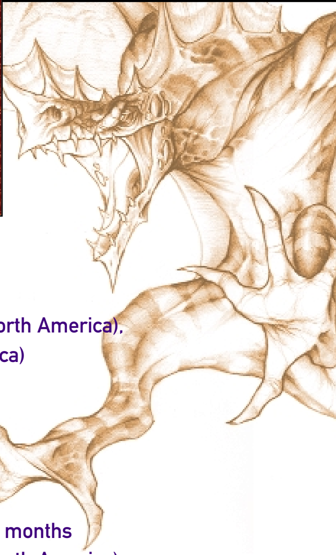
our executive producer, Alan Chaveleh. The game is set in the world of Khaldun, a fantasy setting with strong Persian influences. Players take on the role of a Kohan, one of a race of immortals that were once the ruling power of the world but have had their culture and society destroyed by a series of massive cataclysms. Eternal death is prevented by their immortal nature, and they have begun to resurface. In campaign mode, the player must solve the mystery of the Kohans' destruction and return them to their former glory.

What Went Right

1. Good project setup. One of the good things about our initial corporate infrastructure was that from the start we had the standard support departments (IT, HR, accounting) in place and working efficiently. We were able to hit the ground running, but TimeGate was kept as a privately held, independent corporation. We had the stability of a large corporation and the independence of a small creative studio, with our own custom office space, working hours, and perks.

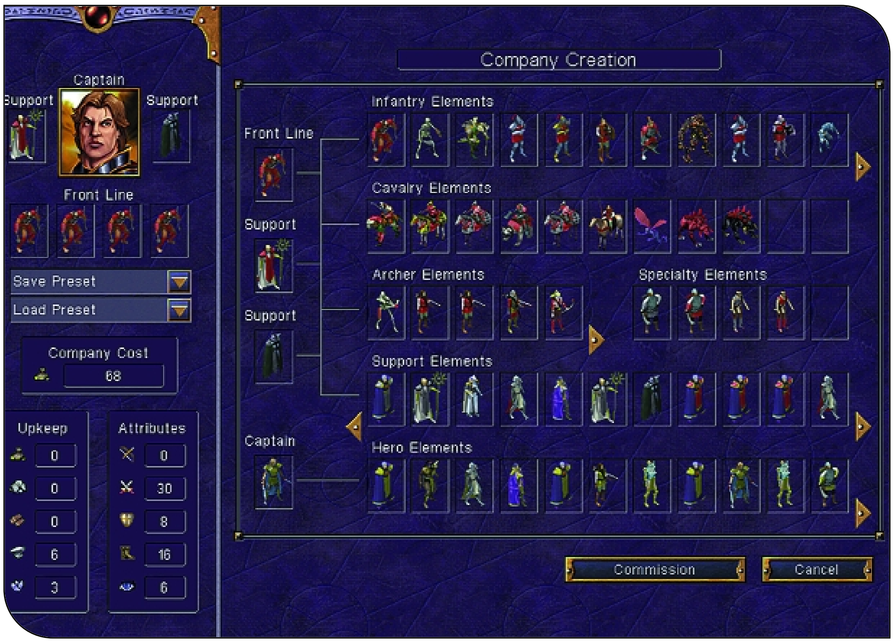
With that solid foundation in place, we started listing the hurdles that we were most likely to encounter. Over time, these evolved into philosophies that we used to keep the project vision in sight. Defining phrases such as "content is king," "we will not be a clone," and "let's keep it simple" were often heard around the office. By adhering to these development philosophies from the beginning of the project, we kept everyone from deviating from our core goals.

Having experienced businessmen at the helm was another great advantage. The founders, who had been doing business successfully for well over a decade, managed all forms of deals, contracts, and negotiations. Being self-funded obviously aided in these negotiations. Ultimately, we landed deals for different territories with publishers that were extremely excited about KOHAN.



GAME DATA

PUBLISHERS: Strategy First (North America), Ubi Soft (Europe, South America) Gamania (Korea, Taiwan) Sunsoft (Japan)
FULL-TIME DEVELOPERS: 12
PART-TIME/CONTRACTORS: 7
BUDGET: $1.8 million
LENGTH OF DEVELOPMENT: 25 months
RELEASE DATE: March 2001 (North America)
DEVELOPMENT HARDWARE: 700MHz Pentium IIIs with 128MB RAM, 20GB hard drives, and GeForce 2s
DEVELOPMENT SOFTWARE: Windows 98 (programming and design), NT (art), Visual C++ 6.0, 3D Studio Max 3.0, Character Studio 2.5, Photoshop 5.0, SourceSafe, Microsoft Office Suite, Microsoft Project, Bugzilla
PROPRIETARY SOFTWARE: TimeGate Editing Tool (TET), TimeGate Art Processor (TAP), Game Integrated World-Editor
NOTABLE TECHNOLOGIES: DirectX, Bink Video, GameSpy, RTPatch
CONTENT: 250,000 lines of C++ (some assembly), 2,000 final resource files, 40,000 words localized
CHIPS AND SALSA ORDERS EATEN: 1,342



TOP LEFT. Magic plays a key role in combat. BOTTOM LEFT. Monsters can become a serious menace if their lairs are overlooked. RIGHT. Players use the company creation interface to customize their army. Each company contains one captain or hero, four front-line elements (such as an archer or beastrider), and two support elements (such as a cleric or magician). The interface was designed to be intuitive and informative.

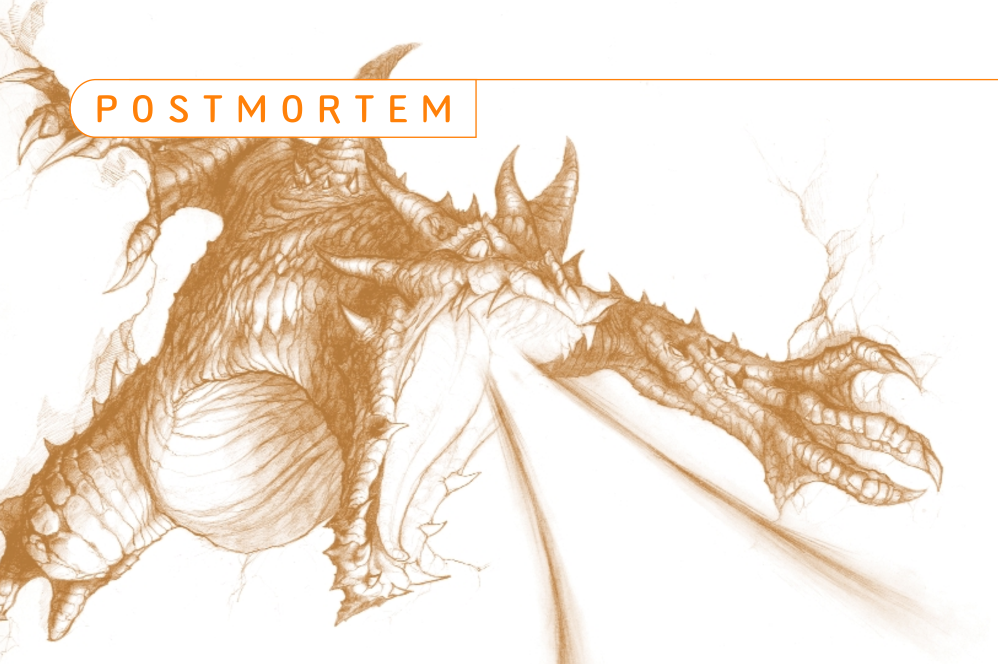
2. Gameplay-driven design. The primary factor to consider if you want to stand out in the RTS genre is good, solid gameplay, and KOHAN's emphasis on gameplay is one of its strongest attributes. With the exception of the interface and program stability, we treated everything else as secondary to gameplay. One of the most common complaints we heard about RTS games was the level of micro-management necessary. These games tend to be very tactical; you micro-manage your economy and each of your military units (even to the point of individually directing them to use special abilities), often resulting in a "click-fest." Few players had time left over for real strategy.

KOHAN's primary focus was to counteract this by streamlining the economy, making the smallest controllable entity the company (a group of seven units), making companies auto-recruit, and introducing a zone system. These are the most critical elements behind KOHAN's gameplay. The upkeep-based economy system was something that was designed after production began — we had already established that we wanted a high-level economic system, but we had to justify the automatic recruiting ability of companies without auto-deducting from the player's gold supply. For companies, we designed rules of engagement and developed a two-layer tactical AI. This AI gives companies the ability to get their units to carry out goals and allows units to use their special abilities, such as spell casting, automatically. The zone management system involved a significant amount of production time to work out and implement, but it allowed us to have terrain-based concepts of visual range, supply, and control. It was all worth it, as raising the primary attention level allows the game to run at a comfortable pace, which enables the player to think in terms of high strategic concepts instead of improving reaction times with the mouse.

KOHAN really tried to bring elements of multiple genres together. We had extensive experience with several turn-based, real-time strategy, and war games. We looked at what worked and what didn't. It was clear that the driving force behind the lifespan of an RTS game is replayability in multiplayer and skirmish games. To this end, we included several different AI players (as well as the ability to create your own), an editor, a versatile random-map generator, and a wide range of options for skirmish/multiplayer games. As a testament to the quality of the gameplay, several members of the development team and public beta team still play the game religiously, and "unbeatable strategies" appear and disappear every week.

3. Minimizing the learning curve. Since KOHAN features several advanced and new concepts for an RTS, we maintained a focus on intuitiveness. Having extensive experience with many popular games, we identified and maintained several interface conventions. The interface emphasizes information grouping, and every action is possible with either hotkeys or the mouse. For basic users, it is important that they be able to play the game with the mouse and that every button have a tool tip — this eliminates frustration for the beginning player (who usually jumps into the game without reading the manual). For advanced users, it is important that they be able to do everything via hotkeys (most of which are indicated in the tool tips) — this eliminates frustration that comes from having to move the mouse to perform an action.

Adapting interface conventions from other popular games was also important. For example, we couldn't understand why



a first-person shooter game would come out that didn't use the standard QUAKE hotkeys, since this is the key set that the majority of FPS players are accustomed to. We made sure that someone could come from playing other popular RTS games and immediately jump into KOHAN. The economic system may require a little adaptation, but the interface does not. A player checking out a demo who can't figure out the interface in five minutes is likely simply to uninstall.

To help players understand the advanced concepts in KOHAN, we included three important features: tutorials, films, and an editor. Tutorials, regardless of whether they are noninteractive, leading, or directing, are absolutely required in a game of any complexity whatsoever. In KOHAN, we take the player gradually through the concepts. We have two tutorial campaigns. In the Basic Tutorial, the player is introduced to required concepts, and in the Advanced Tutorial, the player is introduced to optional concepts (primarily combat-related). Films, recordings of a single- or multiplayer game, have contributed to the game as well. For us, they were useful during the beta phase for tracking down bugs, but since KOHAN's release they have realized their full potential as a scouting tool and a way to enjoy other players' classic battles. Films ensure that early game strategies disseminate throughout the player population, because no one is able to hide themselves or their playing style from the film. Finally, the editor allows players to create new maps and scenarios and to test strategies and concepts.

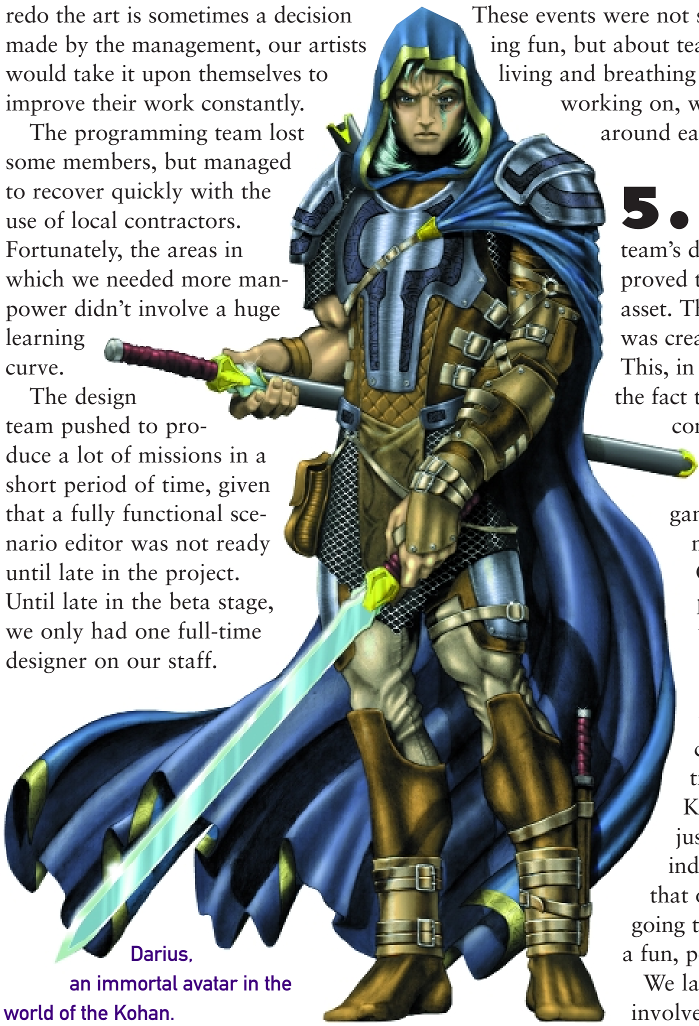
4. Dedicated team. One of the strongest factors contributing to Kohan's high quality and on-schedule completion was the strong team — experience where it counted and plenty of talent.

The art team was primarily new to the gaming industry, but quickly blossomed to produce excellent visuals. They overcame their lack of game industry experience with dedication and an eye for quality. Much of the art in KOHAN was redone several times. While the decision to redo the art is sometimes a decision made by the management, our artists would take it upon themselves to improve their work constantly.

The programming team lost some members, but managed to recover quickly with the use of local contractors. Fortunately, the areas in which we needed more manpower didn't involve a huge learning curve.

The design team pushed to produce a lot of missions in a short period of time, given that a fully functional scenario editor was not ready until late in the project. Until late in the beta stage, we only had one full-time designer on our staff. This required many different people to get involved in aspects of design. This may have detracted from their tasks at hand, but we benefited from it in that there were multiple people that were close to the design.

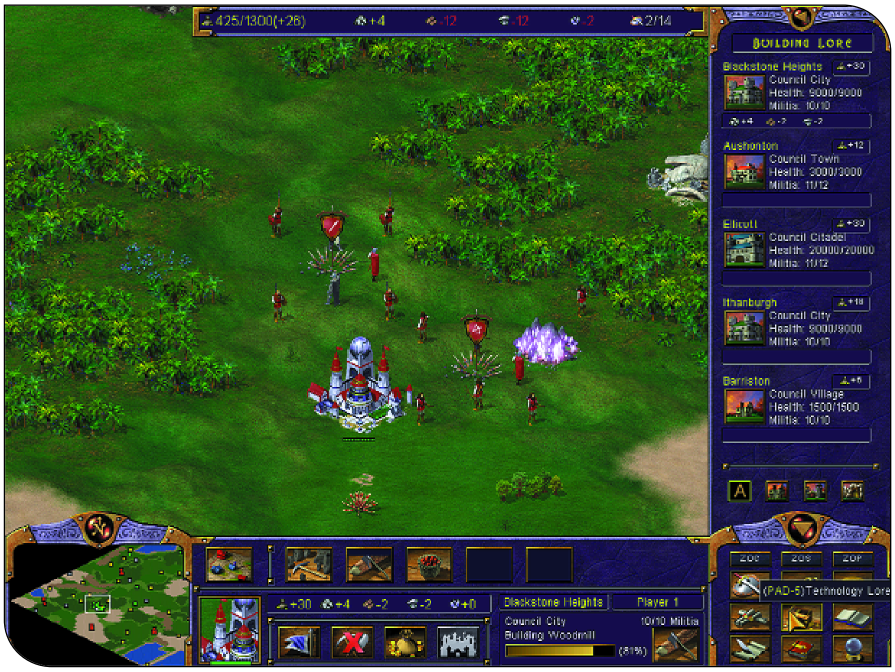
Everyone else was very supportive and made the time when necessary. We had several major milestones that required plenty of extra hours, and the KOHAN team was dedicated to making the game on time and top-notch. The company encouraged this dedication with incentives and morale building. At the beginning of the project we defined a lucrative milestone bonus program and profit-sharing system. In addition, for three straight years (since TimeGate's inception), we've taken the entire team to E3 in Los Angeles, had year-end parties, anniversary trips, and barbeques (we are in Texas, after all). These events were not simply about having fun, but about team building. While living and breathing what we are working on, we still enjoy being around each other.



Darius, an immortal avatar in the world of the Kohan.

5. Strong quality assurance. Our team's dedication to QA proved to be a strong asset. The game engine was created from scratch. This, in conjunction with the fact that our gameplay concept was significantly different from other RTS games out there, made us realize that QA was going to play a huge role in the game's success or failure. TimeGate was a relatively unknown company at the time, and the KOHAN brand was just as new to the industry. We knew that only one thing was going to get us noticed — a fun, polished game.

We laid out a plan that involved six months of



Economic buildings, such as woodmills or quarries, in the cities support the upkeep required by the companies.

beta testing, incorporating a small in-house QA team, two external beta-testing forces (publisher and public testers), and several tools and processes. We implemented necessary features that maximized the amount and type of information we received and our ability to duplicate problems: films, logs, screenshots, save-games, and exception traps. Of course, while all these tools helped, it could have been even better. It was troublesome for users to assemble this information, if they even knew what to provide. An auto-reporting mechanism for crashes and bugs would have helped tremendously.

The day we hit our beta milestone, we received a large number of external beta applications. From this we handpicked a number of people to be our initial group of testers. Our communication with KOHAN's external testers was through the TimeGate beta web site, message board, beta chat room, and our defect-tracking software (the web-based Bugzilla). Beta events were organized and focused. We scheduled weekly multiplayer nights (twice-weekly toward the end of beta) and chat sessions with the external testers. Without test plans, we knew that testing would be haphazard and unfocused, so with each new beta build created (approximately one each week), we provided our external testers with detailed test plans and organized processes for communicating with us.

We became addicted to our own game and would go home from a long day at work and play KOHAN. We were present at all of the beta multiplayer sessions, interacted with our testers on a daily basis, and responded promptly to most bug reports. As a result, our external testers were very active and played a key role in our getting KOHAN out on time and well tested.

What Went Wrong

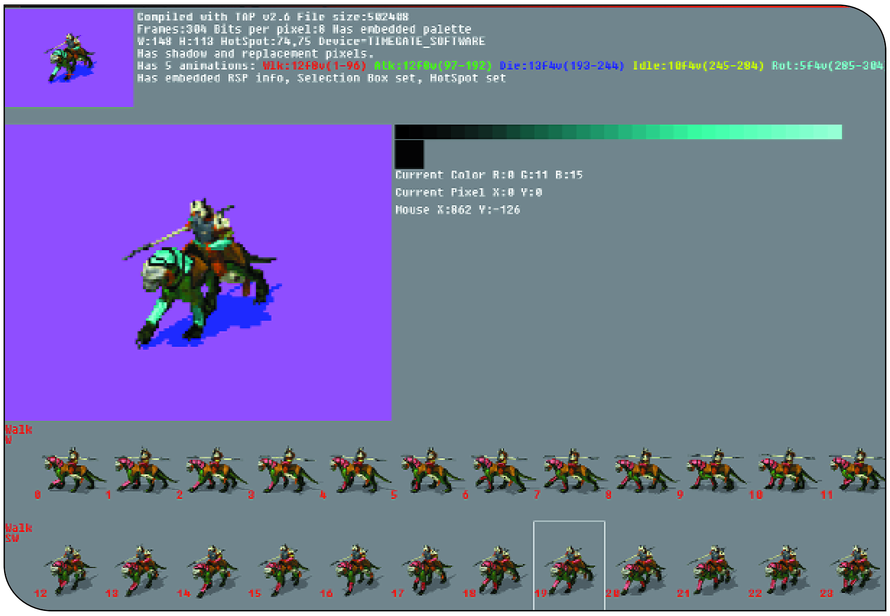
1. Nothing playable until late in the project. If we were to redo KOHAN, one of the first things that we would focus on is having a fully playable version early in the project. Our executive producer had a vision as to what the next evolution of RTS gaming should be. Without something hands-on to show, however, his gameplay concepts and vision were nothing more than a strategy gamer's wish list. We had never played a game that would play the way KOHAN was envisioned. It was difficult getting people to understand exactly how the game would end up looking, playing, and feeling. It was easy to fall into the trap of thinking in terms of other games in the genre. This was beneficial in some cases, such as interface conventions, but was a problem with many of the new gameplay concepts.

The uniqueness of our concepts turned out to be an obstacle for us on two fronts: our development team and potential publishers. Early in the process, the entire development team did not fully understand some of the new concepts. A good bit of early work was completely redone because it did not bring the game closer to making the vision a reality. Working time into the schedule to create a playable version early in the project would have reduced this risk of lost work. However, we did manage to keep the team focused and working toward the original vision through constant design discussions.

The biggest hurdle we encountered was making potential publishers understand the game. No publisher was willing to rely on our word as a new studio that KOHAN was the next big thing. The common response was, "Do you have a demo?" Upon reaching the stage of development that the demo was ready, the interest from publishers peaked, and we ultimately landed the first publishing deal for KOHAN. Creating a demo early thus has the twofold benefit of both getting the entire development team on the same page and having a concept demo for potential publishers. Simply put — schedule time for a demo early in the project.

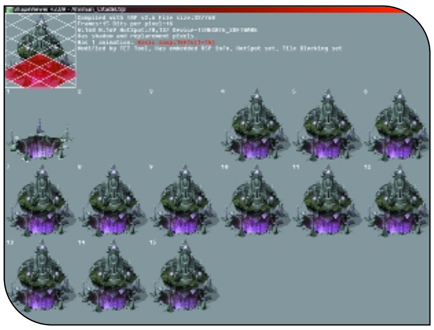
2. Dependency on outside resources. Something that most development houses have to deal with these days is working with outside resources. We believe that, as the game industry matures, outsourcing will become more and more of a viable option for specialty areas of production, such as sound, music, and cinematics. By outsourcing, we thought we could avoid the extra risk of having to invest in specialty employees, expensive equipment, and extra management time and resources.

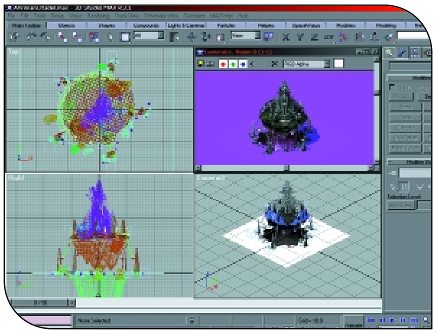
However, subcontracting items that are integral to a project doesn't necessarily reduce risk. Outsourcing has its own risks, risks that were realized in developing KOHAN. We used several contractors for different facets of production, and some



```
Compiled with TAP v2.6 File size:502408
Frames:304 Bits per pixel:8 Has embedded palette
W:148 H:113 HotSpot:74,75 Device=TIMEGATE_SOFTWARE
Has shadow and replacement pixels.
Has 5 animations: Wlk:12F8v(1-96) Atk:12F8v(97-192) Die:13F4v(193-244) Idle:10F4v(245-284) Rot:5F4v(285-304
Has embedded RSP info, Selection Box set, HotSpot set
```

```
Current Color R:0 G:11 B:15
Current Pixel X:0 Y:0
Mouse X:862 Y:-126
```





TOP. The TimeGate Editing Tool (TET) is a custom tool which was created to touch up unit animations. BOTTOM LEFT. Sprite viewer tool. BOTTOM RIGHT. Ahriman's citadel in 3D Studio Max.

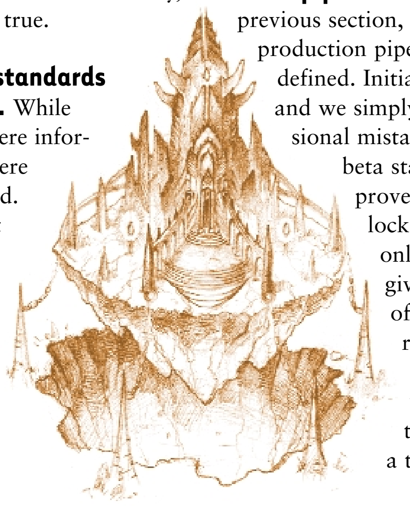
were based in foreign countries. While there was always communication, the time difference often resulted in a day's delay in receiving responses. When we ran into an issue or problem during a workday, we couldn't just pick up the phone and give them a call. This inefficiency would have been avoided had all the work been done in-house. Other resources utilized were more accessible, but while the communication was more efficient with these parties, timeliness of delivering finished goods became an issue. We would receive "finished" work that lacked the quality and polish that we were striving for. This resulted in several back-and-forth sessions that finally yielded something acceptable to all parties, but took much longer than originally planned.

We learned to evaluate closely the costs and benefits of outsourcing specific items and the competence of the contractor. Although at first glance it appears that outsourcing will save you time and money, ultimately it may not be true.

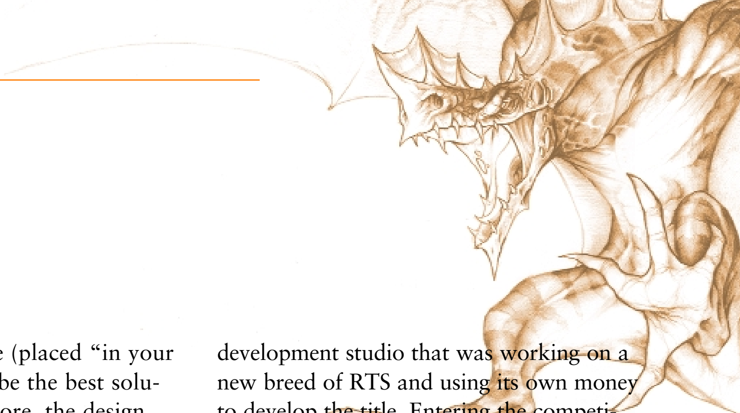
3. Documenting standards and processes. While most of our processes were informally developed, they were not officially documented. For example, a new unit had a particular set of steps it would go through before finding its way into the game. Everyone knew the process (at least with respect to their own involvement), but it was not formalized. Furthermore, we did not have an official checklist for verifying resources from art or maps from design. For example, every sprite had to have a hotspot set, and every campaign map had to have the AI kingdom names set.

In addition, there were several problems with respect to terminology and naming conventions, particularly with respect to the user interface and art files. In the UI, controls were not consistently named, and some interfaces had unintuitive names (for example, the "Multiplayer" interface was really the skirmish game setup interface, while the "Connection" interface was reached by clicking on "Multiplayer"). Without naming conventions, several resources had different names from their respective design resources. There was also no official convention defining what folder a particular type of resource belonged to.

These problems manifested themselves further on in the project in the form of confusion, missteps, and wasted time. The clear solution is to be more aggressive in writing and maintaining documentation and following processes. This requires a disciplined environment, since even the documentation process should be formalized: use a document template and use revision control. Then create an official repository, since these documents must be accessible enough that people get in the habit of referring to them. Once this system is established, define processes, terminology, and naming conventions. Better yet, take the time to write tools that force them.



4. Managing the production pipeline. As we stated in the previous section, several processes in the production pipeline were not formally defined. Initially, this worked fine, and we simply corrected the occasional mistake. However, in the beta stage these mistakes proved costly. We had to lock down the process: only one person was given write access to the official repository of resources for the builds, and that person would verify every resource that went in. This was a time-consuming



process, and still vulnerable to mistakes. Additionally, revision control for binary resources (maps, audio, and art) was done either by hand, or not at all. This was another risky situation, since an error could potentially cost a week or more of redone work.

What was missing was a tool, or set of tools, to handle asset management, both for revision control and for managing the production pipeline (getting an asset from working version to final). Unless the right commercial tool is available, this can be a lot of work, particularly since it should be integrated into third-party tools that the artists and designers use. However, it will save a lot of potential anguish in the long run, more so if it is a project-independent solution.

5. Design document not implemented effectively. The problem was not specifically our design document. The problem was that our design document was not started until production was well under way, and we were pioneering several new game concepts. Not all topics were completely fleshed out prior to reaching them in the schedule. This resulted in changing requirements and feature creep. In turn, those implementing the design were frustrated, schedules were difficult to write and adhere to, and the designer lost a lot of time answering design questions.

Due to the changing requirements, the development of a game feature often resulted in it actually being a prototype. Some of the more complex interfaces were rewritten several times. The schedule, particularly for programming, had to be rewritten several times. We had a significant feature list, and had to make some tough calls about which could be dropped and which could not. Marking a feature as "if possible" was the same as sending it to the guillotine.

These problems spelled doom for the design document. Many things were either worked out when implemented, or worked out in a spontaneous design discussion resulting in notes scratched in notebooks. The initial design document was not complete enough for implementation, so no one read it. It was also not easily accessible (placed "in your face" seems to be the best solution). Furthermore, the design document was not being updated, and, since no one was using it, there were no complaints.

Fixing these problems was a little difficult for a game like KOHAN, which was constantly evolving. It was revolutionary to the genre and difficult to understand, since there was no one game you could relate to and say, "It's like game X." Emphasis must be placed on fleshing out the design document as much as possible during preproduction, and developers must be trained to refer to it. That means keeping it updated, making it easily accessible, and telling everyone to refer to it if the answer is contained therein.

Looking Back, Looking Ahead

The KOHAN project was full of risk from day one. TimeGate was an unknown development studio that was working on a new breed of RTS and using its own money to develop the title. Entering the competitive and overpopulated real-time strategy market was an even greater risk.

In hindsight, we feel that we did a great job calculating and minimizing these risks, and successfully crossing all the hurdles. By working professionally when dealing with people outside the company and by delivering on our promises, our company's image started out on the right foot. Although the KOHAN brand is new, we feel that it is now associated with innovation and quality.

Developing KOHAN was without doubt a learning experience for all of us. However, our success was the result of a dedicated team, an innovative product, and a solid company structure. It was definitely a fun ride. We are already working on our next title, putting all of our newly gained experience and knowledge to use. Going forward, TimeGate's motto will continue to be "Gameplay first."

Return

You're the Boss

There's an epidemic sweeping across the game industry. It's a vast onslaught of gaming tedium that makes an average day of C-SPAN seem like New Year's Eve. I am referring to our boss monster encounters. The elements of the game that should act as the climaxes of our gaming experience are being reduced to boredom and frustration instead of providing the pure gaming bliss that they should.

Now, I'm not naïve. I know that boss encounters are often lackluster because they require special code and art, which in turn translates to money, time, and bugs. Also, they are often moved to the schedule's end, which makes them ripe for gross oversimplification (if they don't get cut altogether). Complex bosses also cannot be reused easily — and resources you can only use once per game are extremely expensive. Still, we can do better.

In this age of beautiful graphics and sound and well-crafted stories, most boss encounters are still "whip out your biggest gun, go mano a mano, and hope you don't die." Which usually devolves into a health meter that moves down too slowly and way too many quick-loads. We've seen minor progress, but this usually entails introducing such strategies as "shoot him when he taunts," "shoot him in the stomach," or if we designers are really clever, "shoot him in the stomach when he taunts." Been there. Done that.

Boss monster encounters should be the emotional highlight of your game, the apex of adrenaline, the part of the game that players describe around the water cooler. And yet, I often hear about people reaching for cheats as soon as they hit the first boss, because we game designers are still making the most basic mistakes. Mistakes such as not putting a save point before the boss encounter. Or not giving any visual feedback of how much damage has been done. Or giving your boss critter insta-kill attacks. Or, God help you, incorporating jumping puzzles into the epic fight.

To all of the game designers out there who are contemplating the boss encounters for their game, I would ask you to do three simple things.

Know the weaknesses of your game. Quite simply, identify what in your engine is fun, and what isn't. Then don't base any boss encounters around the "not fun" parts. The easiest example I can think of is swimming. In many 3D console games, swimming is difficult and unwieldy due to the limitations of the controller. If this is true for your game, then don't put in underwater bosses. Remember, just because you can do something with your engine, doesn't mean you should.

Multiplayer game designers should be aware that latency puts severe limitations on their boss encounters. A superpowerful charging attack that can be easily sidestepped in a single-player game may become impossible to dodge once latency is added to the equation. Timing-related vulnerabilities (for example, shoot him when he taunts) present similar problems.

Server-client games such as DIABLO or EVERQUEST should never have a boss hovering around a teleport-in location, especially if there are long load times. Few things make one want to chuck a CD in the trash quite like dying before your avatar is even visible.

Challenge the notion of a boss monster. Bosses are merely the emotional apex of part of your game. They can be reinvented. Getting out of the space station before it blows or trapping Bossy in an airlock and releasing him into space

continued on page 71

continued from page 72

also have the potential to offer a similar high. SERIOUS SAM did a great job at this by making you kill a thousand little critters instead of one big one. CONKER'S BAD FUR DAY also did a great job by having boss encounters that incorporated the environment.

Also, consider that as designers you do not have to be limited to increasing the challenge level. One can also increase the power of the player. After all, the real point of a boss experience is to provide an emotional high point for the character and player. Consider that an even more potent psychological drive than that of fear is one of self-actualization — the notion that, for a little while, your player can go toe to toe with the big boys.

Have mortals play-test your boss encounter. I buy any shooter I can and typically rank high in any deathmatch game I play. So when I face a boss monster that crushes me the first 19 times I try, my first thought is, "Is it even remotely possible for Joe Wal-Mart to get through this?" And if Joe Wal-Mart gets crushed repeatedly by the first boss, will he finish the game? Or buy the sequel?

If you must have extremely tricky boss monsters, then be sure that you give obvious clues as far as how to beat the monster. Another alternative is putting in some kind of workaround. You don't have to just open a door. You can also give hints at the five-minute mark. Have superweapons appear at the 15-minute mark. Reward the player who can't win, but who can survive.

Boss monster encounters in most games right now are below the quality of the rest of the game — but they don't have to be. Even with the harsh constraints put upon us designers, it is possible to make encounters that elevate the game to new levels instead of making players reach for the cheat codes.

DAMION SCHUBERT | *Damion has been professionally designing games for six years, and is most known for his stints as lead designer of MERIDIAN 59 and the now-defunct ULTIMA ONLINE 2. His current status is unknown, although it is rumored that he is up to no good. He can be reached at damion@zenofdesign.com.*