gd

GAME DEVELOPER MAGAZINE

NOVEMBER 2000

UBM Tech

Return

Sunset, Sunrise

Nearly four and a half years ago, Alex Dunne first appeared in this space. Four and a half years! That's a long time to be doing anything, even more so in our industry, where it's rare to be working on something for longer than two years (well, except for a few games!). To give you a sense of perspective, in that issue of *Game Developer* DirectX 2 was highlighted. At the Computer Game Developers Conference that year, Intel announced MMX, 3Dfx announced the Voodoo graphics chipset, and Apple announced Game Sprockets. Doesn't that seem like a long time ago?

Many things have changed during these years, but you could always depend on Alex and *Game Developer*. Month after month, Alex gave readers insight into our industry, pointed out the shadowy nooks and crannies, and guided developers over bumps such as software patents. Alex didn't just point out industry issues, he created things like the Independent Games Festival, an annual showcase for independently developed games.

It is with sadness that we bid Alex goodbye, but of course he hasn't gone very far. Alex is now the executive producer for our sister publication, Gamasutra.com, and I'm sure he'll bring to that forum the same discerning eye for what's important to you.

Enter, Stage Right

So you may be wondering, who is this guy, and what's he doing here? Hi, I'm Mark DeLoura, the new editor-in-chief for *Game Developer*. Like many of you, I've worked at numerous places, but my primary gig for the past five years has been as software engineering lead at Nintendo of America. While at Nintendo, I was the key Nintendo 64 developer relations person, and I wrote the November 1999 article on rendering Bézier surfaces using N64 microcode. For the last year and a half I've led a team of brilliant software engineers, working on demos and libraries for Nintendo's Gamecube and Game Boy Advance.

My background prior to working at Nintendo includes a heavy dose of virtual reality and an arcade game that never shipped. I spent quite a few years working with virtual reality at the Human Interface Technology Lab and UNC-Chapel Hill, and I co-moderated the Usenet newsgroup sci.virtual-worlds. Then there was the whole networked multiplayer arcade game thing, which sounded like a great idea, and did actually work as long as you ran it on a $250,000 SGI Onyx.

Most recently I organized and edited the book *Game Programming Gems*, which is a collection of programming tips and tricks from 40 professional game developers. (And yes, that would be a shameless plug!)

Engage

Where are we going from here? It's an incredibly exciting and challenging time to be a game developer. In the console space we're faced with four very different platforms to work on: porting games between them is going to be ugly. On the PC side, the range of performance you now must account for demands scalability and hardcore bug-testing. Handheld gaming is launching into completely new areas, with cellular communication, four-player multiplay and console-handheld connectivity. And then there's web gaming, persistent worlds, and the mod community. So we certainly have a lot of material to cover!

It's clear that having a resource to help you make decisions about your games is valuable. We're pleased that one resource you've chosen is *Game Developer* magazine. Our commitment to providing timely, innovative material is unwavering. We plan to broaden our coverage of many types of gaming, as interactive entertainment continues its transition from a niche activity to a truly mass-market art form. These next few years are going to be a doozy!

Mark A. DeLoura

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

ON THE FRONT LINE OF GAME INNOVATION
Game Developer
600 Harrison Street, San Francisco, CA 94107
t: 415.905.2200 f: 415.947.6090 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*
Art Director
Audrey Welch *awelch@cmp.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Daniel Huebner *dan@gamasutra.com*
Jeff Lander *jeffl@darwin3d.com*
Maarten Kraaijvanger *maarten@nihilistic.com*
Advisory Board
Hal Barwood LucasArts
Noah Falstein The Inspiracy
Brian Hook Verant Interactive
Susan Lee-Merrow Lucas Learning
Mark Miller Group Process Consulting
Paul Steed Independent
Dan Teven Teven Consulting
Rob Wyatt The Groove Alliance

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
Account Manager, Western Region, Silicon Valley & Asia
Mike Colligan e: *mcolligan@cmp.com* t: 415.947.6223
Account Manager, Northern California
Susan Kirby e: *skirby@cmp.com* t: 415.947.6226
Account Manager, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 415.947.6224
Sales Representative/Recruitment
Morgan Browning e: *mbrowning@cmp.com* t: 415.947.6225

ADVERTISING PRODUCTION

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

CMP GAME MEDIA GROUP MARKETING

Senior MarCom Manager Jennifer McLean
Strategic Marketing Manager Darrielle Sadle
Marketing Coordinator Scott Lyon

CIRCULATION

BPA INTERNATIONAL
Game Developer magazine is BPA approved

Group Circulation Director Kathy Henry
Director of Audience Development Henry Fung
Circulation Manager Ron Escobar
Circulation Assistant Yumi Sato
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

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

CMP MEDIA MANAGEMENT

President & CEO Gary Marshall
Corporate President/COO John Russell
CFO John Day
Group President, Business Technology Group Adam Marder
Group President, Specialized Technology Group Regina Ridley
Group President, Channel Group Pam Watkins
Group President, Electronics Group Steve Weitzner
Senior Vice President, Human Resources Leah Landro
Senior Vice President, Global Sales & Marketing Bill Howard
Senior Vice President, Business Development Vittoria Borazio
General Counsel Sandra L. Grayson
Vice President, Creative Technologies Johanna Kleppe

CMP

WWW.CMPGAME.COM

Return

FRONT LINE TOOLS

WHAT'S NEW IN THE WORLD OF GAME DEVELOPMENT | *jennifer olsen*

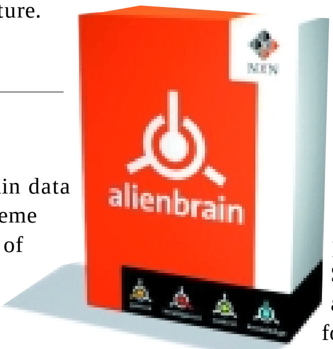
ODDWORLD: MUNCH'S ODDYSEE, Oddworld Inhabitants, All Rights Reserved

NETIMMERSE 3.1

A new version of NDL's NetImmerse game engine has arrived, featuring additional plug-in support for real-time previewing, better Playstation 2 performance, and new animation features. In addition to improving the existing MaxImmerse plug-in for 3D Studio Max, which now features real-time previewing for PS2, NetImmerse 3.1 adds two more plug-ins: MyImmerse, a real-time previewing plug-in for Maya 3 with multi-texturing support, and MultImmerse for Multigen Creator 2.3. Other new features include an optimization library and keyframe verification tools, aimed at helping programmers to write better-performing code, plus support for the Miles Sound System on PC, as well as better animation blending and control over sequences. NetImmerse currently supports Windows, MacOS, Linux, and Playstation 2, with Xbox support planned for the near future.

NETIMMERSE 3.1 | NDL | www.ndl.com

ALIENBRAIN TURNS UP THE POWER



NxN has introduced a new, more powerful server for its Alienbrain data management system for game development, the Alienbrain XTreme server. Largely a response to the rapidly increasing content demands of next-generation games, Alienbrain XTreme is optimized for projects with well over 100,000 files and more than 20 developers, numbers which are increasingly characteristic of today's game development projects. The Alienbrain XTreme server allows users to widen the scope of the projects managed by the Alienbrain system, handling larger data sets and providing simplified backup options plus faster startup and shutdown times.

ALIENBRAIN XTREME SERVER | NxN | www.alienbrain.com

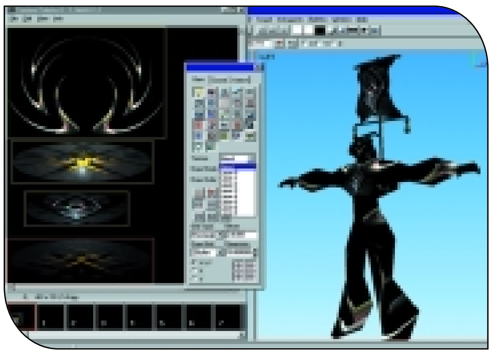
INTRINSIC ALCHEMY BETA RELEASE

Intrinsic Graphics has announced the first publicly available version of their Intrinsic Alchemy game development platform, aimed primarily at simplifying simultaneous cross-platform development, something many game companies are pondering for current and upcoming projects as the next generation of game platforms emerges. Intrinsic Alchemy works by separating a game's processing blocks from the application structure and data flow of the underlying platform, and using an Application Graph as the identity of processing blocks and the data connectivity between them. A developer's compiled C and C++ functions can then traverse the Application Graph at run time and execute the processing blocks on the intended platform.

The current beta release supports Windows, Playstation 2, and Sony's GS Cube, with licenses available for single or multi-platform use. Support for Gamecube, Xbox, and unspecified STB systems is planned for upcoming releases.

INTRINSIC ALCHEMY | Intrinsic Graphics | www.intrinsic.com

MULTIGEN CREATOR 2.4



Multigen-Paradigm has updated its Multigen Creator 3D modeling program with a major new release. Creator 2.4 adds multi-texturing of up to eight textures per polygon, the Irregular Mesh polygon-generation algorithm for terrain creation, a new unified project management interface for Terrain Pro, plus an updated OpenFlight file format, giving access to multi-texturing and terrain T-mesh nodes for irregular mesh databases. Creator 2.4 is available for Windows 98/NT/2000 and IRIX 6.2 or higher.

MULTIGEN CREATOR 2.4 | Multigen-Paradigm | www.multigen.com

SAFEDISC 2.0 TRIES TO THWART HACKERS

Macrovision has released a new version of its SafeDisc PC CD-ROM copy protection system for publishers, developers, and replicators. SafeDisc 2.0 includes a major overhaul of the SafeDisc code architecture, in an effort to circumvent automated hacking tools and provide additional encryption to delay hacking. In addition, Macrovision has added new digital signatures to improve resistance against copying, and manufacturing assurance that SafeDisc masters will only be mastered on SafeDisc-enabled production lines. SafeDisc includes authenticating digital signature(s) embedded on a CD-ROM, an encryption wrapper, and an antihacking technology that secures the CD-ROM executable and prevents copying to standard CD-R drives. SafeDisc 2.0 is available now for Windows 95/98/NT/2000/ME.

SAFEDISC 2.0 | Macrovision | www.macrovision.com

Return

INDUSTRY WATCH

THE BUZZ ABOUT THE GAME BIZ | *daniel huebner*

Nintendo Announcements

Nintendo used Spaceworld as its venue to finally unveil its next-generation console and handheld. The system formerly known as Dolphin has been rechristened the Nintendo Gamecube. The Gamecube is built around a 405Mhz copper central processor from IBM, an ArtX-developed graphics processor, and 40 MB of memory. As many had speculated, the Gamecube will use an 8cm, 1.5GB proprietary optical disc. Gamecube will not feature a harddrive, instead utilizing flash memory cards. Nintendo also revealed details about the upcoming Game Boy Advance. The new handheld is powered by a 32-bit ARM processor and features a reflective TFT screen that is 50 percent faster than the current Game Boy Color. Game Boy Advance ships in Japan in March 2001 at a price of around $90; it will arrive in North America and Europe next July. The Gamecube is expected in Japan in July 2001, and should make its way to the rest of the world by October of the same year.

The Nintendo Gamecube and its controller.

New FTC Report Calls for Stricter Rules for Selling Games

A much-anticipated government report titled "Marketing Violent Entertainment to Children: A Review of Self Regulation and Industry Practices in the Motion Picture, Music Recording and Electronic Gaming Industries" strongly criticizes the entertainment industry for its marketing practices. Written by the U.S. Federal Trade Commission, the report recommends that more should be done to address how media markets violent entertainment to children, and calls for stricter measures at the retail level to prevent minors from purchasing violent games. In anticipation of the report, K-Mart and Wal-Mart made moves to restrict sales of mature games. K-Mart announced that the chain will refuse to sell M-rated mature game titles to anyone under the age of 17, and Wal-Mart has announced that its stores will enact the same policy. More retailers are expected to follow suit, and companies such as Electronic Arts are voicing support.

Game Ban Suit

Two game industry groups are suing to overturn an Indianapolis law barring minors from certain coin-operated videogames. The Indianapolis law allows coin-operated games featuring graphic violence or sexual content to be played only by those over 18, and such machines must be affixed with warning labels and kept out of the view of minors. Indianapolis Mayor Bart Peterson, county prosecutor Scott Newman, and local police officials are named in the suit. The American Amusement Machine Association and the Amusement and Music Operators Association (AMOA) want a judge to grant a temporary restraining order to keep the law from going into effect on September 1. "We are on the edge of a slippery slope, and our industry has been forced to litigate to protect core constitutional rights," explained AMOA president Frank Senisky.

Nvidia Sues 3dfx

Graphics chipmaker Nvidia is suing competitor 3dfx for patent infringement. The suit, filed in a Northern California district court, involves five Nvidia patents that the company believes have been infringed upon by 3dfx products. Nvidia is asking for an injunction to prevent 3dfx from selling its Voodoo 3, Voodoo 4, and Voodoo 5 product lines, and is also seeking monetary damages. 3dfx believes that Nvidia is bringing the suit solely in order to force a settlement of the multi-texturing patent suit 3dfx filed against Nvidia in 1998.

Ubi Buys Red Storm

Paris-based Ubi Soft Entertainment is purchasing North Carolina's Red Storm Entertainment. Red Storm will become a wholly-owned subsidiary and brand of Ubi Soft. Though sales and distribution functions for the two companies will be integrated, Red Storm will continue to operate independently continuing to focus on its ongoing projects as well as continuing its association with Tom Clancy, a license that has been extended as part of the Ubi Soft deal. Other developer buyouts are also afoot, with Champaign, Illinois–based Volition becoming a part of THQ. The publisher bought Volition, a spin-off from DESCENT developer Parallax Software, in exchange for stock and the assumption of $500,000 in net liabilities. The deal, which also includes Volition's Parallax Online game matching service, was finalized on August 31. Volition is currently working on SUMMONER and RED FACTION for PC and PS2.

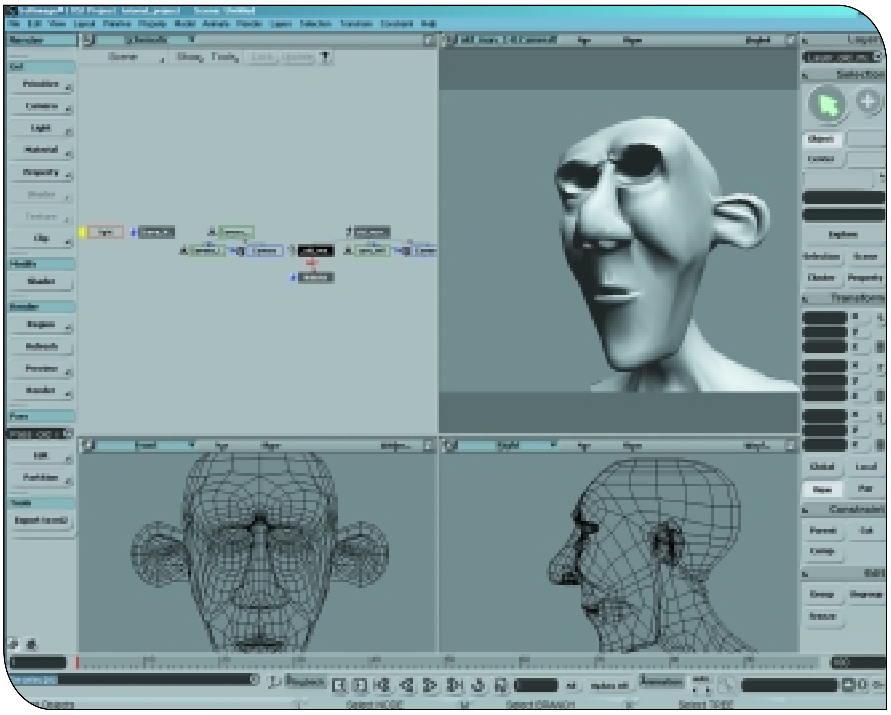
Sega Price Cut

Sega cut the price of its Dreamcast in an effort to highlight the console's online gaming capabilities in the wake of the Playstation 2 launch. The company has set a price point of $149 for the console, an expense potential buyers can wipe out entirely by participating in Sega Net's $150 Internet service rebate scheme. Sega is also tempting consumers by showcasing anticipated online multiplayer games NFL 2K1 and QUAKE 3: ARENA for Dreamcast.

UPCOMING EVENTS CALENDAR

COMDEX FALL
Las Vegas Convention Center
Sands Expo & Convention Center
Las Vegas, Nev.
November 13–17, 2000
Cost: variable
www.key3media.com/comdex/fall2000

Return


The XSI Interface with an imported polygonal model from Softimage 3D.

Softimage XSI 1.0

by david stripinis

Softimage. For years those three little syllables rolled off the tongues of 3D artists everywhere with wonder. But then something happened. 3D Studio became 3D Studio Max. PowerAnimator became Maya. And Softimage . . . well, Softimage remained the same. Of course, it went through incremental updates, with feature additions and interface enhancements, but the core remained the same solid foundation on which hundreds of games and movies have been produced. While not necessarily a bad thing, the "buzz" was with Max and Maya. But there was this word whispered in quiet corners of studios and art departments around the world: "Sumatra." And it wasn't just a request for the intern to pick up some Starbucks. Sumatra was Softimage's oft-delayed next-generation 3D production tool, and it has finally arrived in a big way. Sumatra, now known by the somewhat less exciting name of Softimage XSI, builds on its strong heritage, and adds exciting new functionality. Drawing on the expertise of Softimage while adding the talents of Avid's engineers, XSI is definitely going to turn a few heads.

When you first start XSI, you realize it is not your average Windows application. It appears as if Avid's Macintosh legacy has heavily influenced the interface. The only standard Windows features I could see were the title and menu bars. Even the file dialogs use Softimage's Unix/Mac-blended UI design. This unique UI did cause one minor technical problem: as Softimage starts, it sizes itself to your current resolution — ignoring the Windows Taskbar. Setting the Taskbar to Autohide seemed to be the best solution. The UI itself is divided into five main areas. Taking up the most real estate are the workspace viewports. To the left is the Toolbar area, which is divided into Modeling, Animation, and Rendering tool sets. This is where you readily access most of your commands. The top is the main menu bar, where you access all the general functions of the program, as well as the toolsets of the Model, Animate, and Render menus without having to switch modules. The bottom contains all the timeline and command line functions. Strangely, this is also where you access all the animation editing tools. I say strangely because it was literally the last place I looked for them while going over the program the first time. Finally, the right hand of the screen contains the Main Command Area. This ominous-sounding control panel contains everything from layer controls and selection filtering to transformation and grouping tools.

One very unique feature I have come to love is the concept of "sticky" hotkeys. By quickly tapping a key, you enter into a mode or tool. Tapping the key again returns you from that mode. In addition, if you simply hold a key down, you remain in that tool or mode only as long as that key is depressed. This is a wonderful enhancement to standard hotkeys, and I find myself missing it when I'm working outside of XSI.

I can't say I liked the interface as a whole, though. It obviously tries to retain the feeling of previous versions of Softimage while adding workflow enhancements and access to new tools. Anyone who has become accustomed to the standard Windows interface will most likely have trouble. However, artists who have never ventured outside of Softimage should have no trouble whatsoever.

The other UI element you will encounter most often is the Property editor. The Property editor offers access to every bit of data available on any object, and is also

DAVID STRIPINIS | *David is director of animation at Factor 5. Feel free to contact him at david@factor5.com.*

CLARIFICATION: In our two-part review of physics engines (September and October 2000), we neglected to mention that the version of MathEngine we used was an alpha. As was mentioned in the second part of our review, MathEngine is now shipping alpha version 0.0.5. Havok provided us with Havok Evaluation version 1.2. Ipion supplied the Ipion Virtual Physics SDK version 1j9.

one of the most convenient ways to access the construction history on an object, called the Operator Stack. Everything you do to an object, from applying a deformer to moving a vertex creates an operator, and can be easily accessed and keyframed through the Property editor.

Modeling

Softimage has an extensive and robust set of tools for modeling NURBS surfaces. This is a welcome change from the days of Softimage 3.x, where the NURBS modeling features were decidedly subpar. In particular, I found the Curve Net and Continuity Manager to be particularly powerful and useful tools in creating seamless, detailed organic surfaces. In addition, surface fillets, surface blends, bi-rail surfaces and four-sided surfaces provide a complete suite of tools for creating anything you may desire — using NURBS.

This is where we come to Softimage XSI's largest and nearly fatal flaw. You cannot create polygonal objects of any real use with the program as it now stands. While you have access to a variety of polygon primitives, the only way to edit them is pulling on vertices or using deformers. This is, without a doubt, unacceptable for in-game model creation. This is the main reason for the inclusion of Softimage 3D, which has a full suite of polygon editing and creation tools. In particular, I found their UV editing tools to be quite excellent. Luckily, Softimage XSI can easily import Softimage 3D scenes completely intact, including any polygonal objects.

While using Softimage 3D is quite a powerful solution to XSI's polygonal shortcomings, forcing game artists to learn two programs to create assets is a hindrance to Softimage XSI gaining ground with companies dedicated to Max, Maya, Lightwave, or Mirai. To artists already using Softimage 3D, this should be no big deal, and may actually help in the process of switching to Avid's next-generation offering.

Animation

To most 3D artists, the word "Softimage" is synonymous with animation. Softimage first introduced most of the indispensable tools and techniques for modern computer animation. From IK to constraints, function curves to dope sheets, Softimage's touch is felt industry-wide in every package. They hope to continue that tradition of innovation and revolution with nonlinear animation.

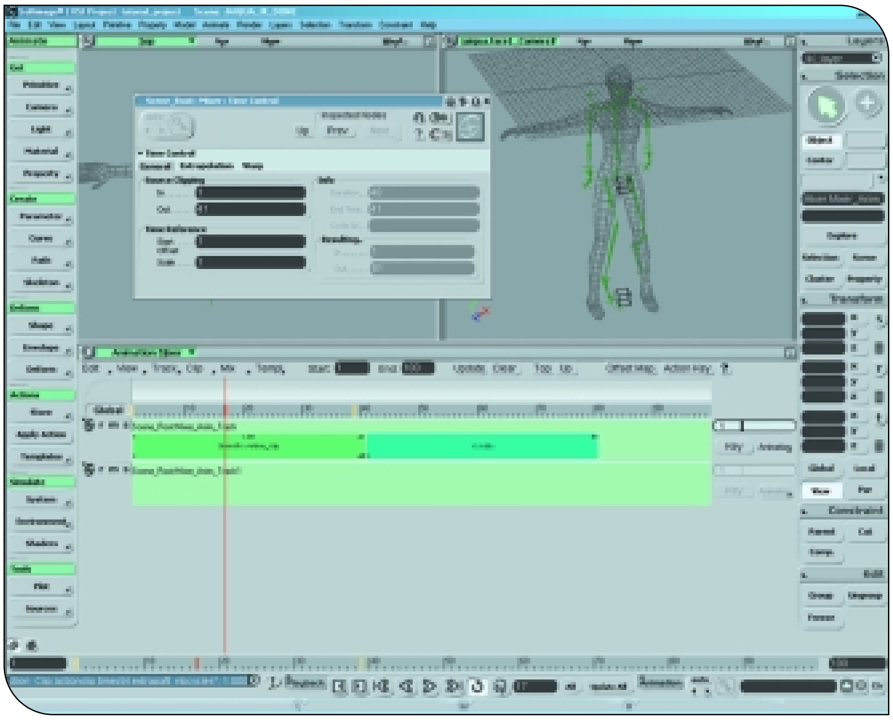
Nonlinear animation is nothing new to game artists. We've been doing it for years without even realizing we were revolutionizing the very process of animating. To the uninitiated, nonlinear animation (NLA) is the process by which different motion assets can be strung together, blended, combined, and generally messed with in every conceivable combination, all in a nondestructive fashion. You access XSI's NLA features through the Animation Mixer. This interface is similar to nonlinear editors such as Adobe Premiere or (this should come as no surprise) Avid Media Composer. The simplest application of this technology is to use in-game animation assets to string cutscenes together quickly, and to keep the motions "in-character." But this is missing the true power of NLA.

By overlaying and blending multiple clips, an artist can create seamless transitions between separate actions, or layer animations on top of each other. Once the possibilities of these functions are fully realized, animators will be drooling to get their hands on these tools.

For instance, no longer will animators slave away trying to get all the start and end poses of their animations exactly the same. You can simply save the pose as an Action Clip and bring it in at the beginning and end of your cycles to get a perfect match-up. The blending functions will also greatly help in hand-creating transition animations.

NLA is a truly revolutionary tool and will be welcomed by artists everywhere. It will be interesting to see how this technology develops in the coming years. XSI already offers a robust NLA tool suite, and looks to be an industry leader in this exciting technology.

Softimage XSI uses very solid IK routines. What's amazing is that XSI builds the IK directly into the skeletal chains, which makes rigging skeletons quick and simple. I'm a control freak, and don't like being removed from the rigging process, but I can see the appeal for many artists.


The Animation Mixer. XSI continues Softimage's role as innovator with its nonlinear animation system, which is pictured above.

✶✶✶✶✶	excellent
✶✶✶✶	very good
✶✶✶	average
✶✶	fair
✶	don't bother

Because the IK is built into the skeletal chains, an animator can use forward kinematic techniques to pose the skeleton without having to do any wacky constraint tricks. Simply select the bone and rotate it. If you're an animator, I know you're smiling with glee right now.

Scripting

Like Max and Maya, XSI incorporates scripting; unlike Max and Maya, however, XSI does not rely on a proprietary language. Instead, it relies on ActiveX, so anything supporting ActiveX will work, including VBScript, Jscript, PerlScript, and Python. These are proven tools and will surely be exploited by enterprising artists.

Rendering

Softimage has deeply incorporated Mental Images' Mental Ray renderer into XSI. While I know to many games artists rendering is completely pointless, I would not be doing this program justice without mentioning this aspect. After all, for many games today, prerendered video sequences are a major selling point.

The images produced with Mental Ray are of astounding quality. The system is renowned in the film industry for its crisp raytracing and wonderful volumetric effects. Any company relying on rendered cutscenes or images for their game will find the possibilities Mental Ray offers them intriguing.

The Verdict

I now have to give Softimage XSI a star rating. If all you are interested in is creating a prerendered cinematic, then give XSI a long look. But its lack of polygon tools hurts this product too deeply for me to give it a recommendation. It has potential, but there really is no excuse for offering a product without polygon tools at this stage. I look forward to seeing where Avid takes Softimage in the near future.

SOFTIMAGE XSI ✶✶½

STATS
SOFTIMAGE (A DIVISION OF AVID TECHNOLOGIES)
Montreal, Quebec (514) 845-1636
www.softimage.com

PRICE
Softimage XSI Essentials: $7,995
Softimage XSI Advanced: $11,995

SYSTEM REQUIREMENTS
For Windows 2000/NT (SP 4 or higher); workstation with Intel Pentium or higher; compatible OpenGL-accelerated graphics card with a minimum 8MB RAM; 128MB RAM required, 256MB RAM recommended; 195–360 MB disk space. N32 libraries and patches; SGI workstation with MIPS R10000 or higher processor; 128MB RAM required, 256MB RAM recommended; 645MB disk space. Both platforms require a three-button mouse and CD-ROM drive.

PROS
1. Excellent animation tools.
2. Nonlinear animation tools.
3. Powerful NURBS modeling.

CONS
1. No polygon tools.
2. Unconventional interface.
3. No polygon tools.

Trigger

a.k.a. Trap, Egg, Timer, Condition/Response

Problem

Games often involve intricate puzzles and sequences of dependent events. For example, players sometimes must navigate a series of obstacles or traps, using or avoiding switches, secret doors, mines, guards, and booby traps. Similarly, a plot element in a game may need to happen once a certain character is dead or after the player has acquired a specific item. Coding such traps and event-chains directly in a high-level language such as C or even in a custom embedded scripting language is often tedious because it usually involves numerous arbitrary constants that clutter the code and change frequently as the game design is tuned.

Solution

A special in-game object called a "Trigger" is created that monitors conditions and responds to them. Designers can then create triggers, place them, and change their conditions and responses in the game's design tool. Programmer time is not required after the library of Trigger options have been created. Conditions can be geometric, time-based, or event-based. Responses can include creating or notifying objects, or other actions such as aggregating and notifying Triggers.

Using a separate Trigger object (as opposed to hard-coding conditions and responses), decoupling conditions and responses, and developing good user interface in the editing tool can create a powerful visual programming language for designers.

Examples

Conditions. There are many types of geometric conditions, like "Has the player entered the cave?" or "Is the player within a specified radius of the altar?" Time- and event-based conditions are commonplace as well, including, "Has the player been in the dungeon for five minutes?" and "Is the boss-monster dead?"

Responses. Responses vary widely. For example, "Fire poison darts at player," "Wake up boss-monster object," "Notify all objects linked to me," and "Notify all enemy-AIs within a given radius." Triggers made from multiple conditions and responses are possible and desirable: "If a player enters the room, picks up the amulet, and stays for more than two minutes, close the door and fill the room with gas."

Issues

Editing. The design tool's user interface must be developed to allow creation, sizing, placement, and other editing functions to be applied to the trigger without programmatic interfaces. Designers may want multiple interfaces to the Trigger objects, including the editing interface and access via a scripting language, if present. This UI can become very complex.

Debugging. Complicated sets of Triggers broadcasting and reacting to one another can be very difficult to debug. It is helpful to construct a break-point or single-step interface that allows the designers to monitor the progress of the system as it executes. Visualization tools, such as flying broadcast tags or flashing the send/receive pairs, are also useful.

Predicates. As more sophisticated traps are constructed with Trigger implementations, it often becomes necessary to include more traditional programming constructs. For example, the condition "only if first time to enter the cave" might be needed on a specific proximity Trigger. The need for such specific predicates is usually only discovered during puzzle development, and therefore such conditions are often added piecemeal to the Trigger-edit UI until it becomes cluttered with obscure options. More general implementations may allow scripts to be associated with the Trigger and edited in the same UI without need for recompilation. However, it is also frequently true that a small and well-engineered set of simple conditions, responses, and global flags will suffice to create a large variety of traps. In this case, exceptional situations might be more easily made in the native programming language, avoiding a complicated scripting solution that would be used only rarely.

Performance. Triggers that perform expensive and frequent condition evaluation can be performance problems. Geometric Triggers that query proximity in 3D space could make use of a Spatial Index. Scheduling Triggers for condition testing can be an important part of optimization.

Related Patterns

Uses Spatial Index. See the Patterns Database mentioned below. Conditions and responses are called "Triggers" and "traps" respectively at some game companies.

Uses, Credits, and References

Triggers are common in many games. They date back to the earliest console computer games. One reviewer mentioned that THE PIT, a 1982 Atari game, contained the first Trigger he noticed. The author was heavily influenced by several proponents at Origin Systems, especially Gary Scott Smith and Herman Miller.

CHRIS HECKER | *Chris Hecker (checker@d6.com) is editor-at-large of* Game Developer.
ZACHARY BOOTH SIMPSON | *Zack is the former director of technology at Origin and Titanic.*

Now It's Up to You!

This column depends on your contributions! Send your patterns and idioms to us at patterns@d6.com. If we publish your pattern, we'll give you recognition in print and $100!

Haunted Trees for Halloween

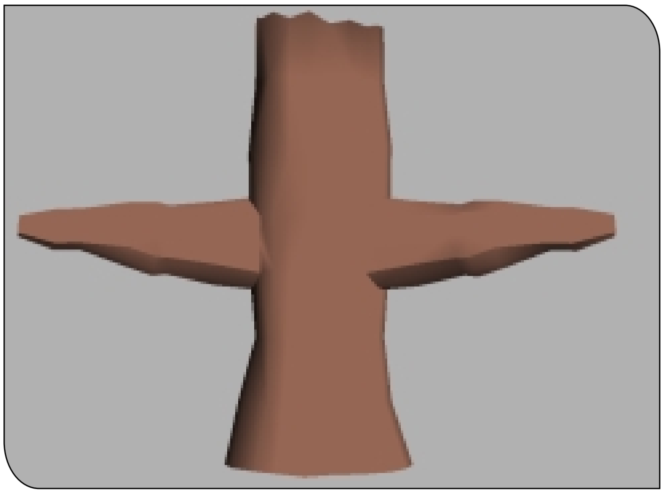


FIGURE 1. The tree mesh, a good starting point for a spooky tree.

As a child, camping in the woods always fascinated me. The stillness at dusk and the total lack of light pollution that made the nights completely dark brought me a mixture of awe and anxiety. When I was a camp counselor during the summer, the camp staff would play on those feelings to enhance the experience for our elementary-school-age campers. We would tell stories of great horned owls that flew silently through the night and were large enough to carry a child away. We warned them to stay away from the oak grove at night, because the trees came alive, grabbing hikers and pulling them into their evil core. To support this myth, during our day hikes we would point out twisted knots in the trees that looked like faces in agony. Occasionally, on the way to the campfire at night, several of us would split off into the woods, to make the forest "come alive." Yes, we were demented little weasels.

We were not the only ones to play up the myth and mysticism surrounding great old trees. History's tales have always been filled with haunted forests and enchanted trees that both heal and harm. The tree has become a paradoxical symbol of history, wisdom, birth, and death. A sapling grows from the rotting flesh of its ancestors. The adult tree silently watches the world around it.

It's no wonder this symbol has been used throughout the history of storytelling. Directors of Westerns love to evoke the hanging tree as an image of despair. I remember watching *The Legend of Sleepy Hollow* on *The Wonderful World of Disney* as a child. The way the trees looked and moved gave me more than a few nightmares. And now kids, and adults, have been enjoying the lore surrounding the Whomping Willow in the *Harry Potter* series. Even games are not immune to such symbolism. Anyone else remember fighting the haunted trees in DUNGEON MASTER 2?

So, in celebration of the Celtic New Year, Samhain, this month I am going to build on last month's 3D paint system and pull in some more technology to create a haunted tree.

Planting the Seeds

The first step is to model the tree. I just built something that I thought looked like a spooky old tree with limbs that could be used as arms. You can see the tree mesh in Figure 1.

The next step is to take the old digital camera outdoors and find a nice tree to capture, and massage it in a 2D paint program to make a texture that will work. Using the 3D paint capabilities of last month's program, I can touch up any areas that need some work. Actually, to make a confession, I just use the 3D system to mark places I want to adjust, then go back to a 2D paint program. I haven't had time to add different brushes, an airbrush, smudge, or other nice tools to the 3D paint system. It is pretty useful just for marking the exact 3D spot on the texture.

Refoliating

The texture makes a rather nice looking bare tree. Nevertheless, I really want to add some other branches and leaves to the tree. Creating 3D trees has been the bane of many art directors in computer games. Artists have tried everything from the simple billboard tree texture to the pine-tree teepee. Thankfully, computers are getting a lot faster, and we can devote more graphics bandwidth to improving the objects in our scenes.

I was really impressed with the organic look Disney achieved in their recent animated feature *Tarzan*. Through the use of 3D strokes painted inside the 3D environment, Disney created a very compelling and complex jungle environment.

The process of painting 3D content directly on the geometry in a scene is very similar to the 3D texture painting operation I described in September ("Art and Intelligence: 3D Painting"). I need to detect the point where the user clicks on the object. However, for this application, I need to figure out the actual 3D coordinate in the scene where the user clicks, rather than the UV coordinate.

There are many methods for calculating a clicked coordinate in a 3D scene. For example, I could look at the transformed coordinates for each triangle and determine it from that information. This method doesn't take advantage of the features of the rendering hardware such as the Z-buffer. Instead, I'm going to use an image-based method that is similar to the 3D paint technique, and use the graphics hardware to make the calculation easier.

JEFF LANDER | *When not scaring the tar out of little children, Jeff can be found in his deep, dark cave at Darwin 3D. Send him your favorite spooky story at jeffl@darwin3d.com.*

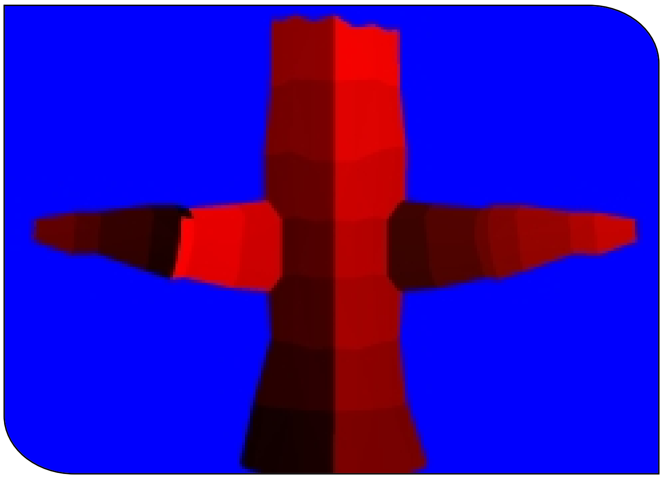


FIGURE 2. Render with polygon number encoded.

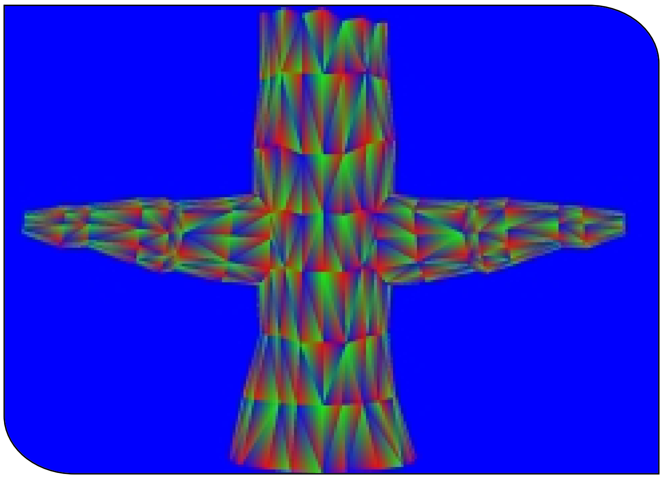


FIGURE 3. Rendered tree with vertex encoding from Listing 1.

The first thing I need to determine when I click on a mesh is what polygon I have clicked on. I can encode the polygon number into a color and render the mesh with that color for the entire polygon. Here is a sample color encoding scheme in OpenGL that will handle meshes of fewer than 65,536 polygons, which is more than plenty for my needs.

```
glColor3ub(polyNum % 256, polyNum / 256, 0);
```

The red component is the polygon number modulus 256, and the green component is the polygon number divided by 256. When I render the scary tree using this technique, I get the image in Figure 2. Just like this 3D paint application, the user never sees this rendering. It is drawn to the back rendering buffer whenever the viewpoint is moved and then stored to a memory location I created for it. The OpenGL commands to copy this render off my temporary buffer are:

```
glReadBuffer(GL_BACK);
glReadPixels(0,0, m_ScreenWidth,
  m_ScreenHeight, GL_RGB,
  GL_UNSIGNED_BYTE, m_TriCountBuffer);
```

Now when I click anywhere on the screen, I can consult this buffer and find out what polygon was hit. That will give me the 3D coordinates of the vertices in that polygon, but not the exact coordinates where I hit.

To determine the exact location, I am going to create another buffer. This time, however, I am going to encode a color at each vertex. My mesh is a triangle mesh, meaning it has three vertices per triangle. Vertex 1 will be colored red, Vertex 2 colored green, and Vertex 3 colored blue. The OpenGL code looks like Listing 1.

The color interpolator on the graphics card will create a smooth blend between each color. The resulting image (Figure 3) looks kind of wacky, but it is very useful. I get the color at the picked position from the vertex position buffer that has the colors encoded as RGB values from 0.0 to 1.0. With these two buffers rendered out, I have everything I need to find out exactly where I clicked. I take the location of the click and check the two buffers. That gives me a triangle number, `triNumber`, and an RGB value for the vertex number, `vColor`. The pseudocode for then calculating the clicked point is shown in Listing 2. Actually, with a Vector class in C++, it would be fewer lines than what is shown, but not as easy to read.

FIGURE 4. A couple of leafy textures.

When the Leaves Turn

Now that I have a fast and easy way to select an exact 3D position on a model, I need to hang something on the tree. I could actually create geometry and add it to the mesh, but I would really like just to attach the leaves to the tree. I am going to treat the leaves more like a particle system. The tree structure has a list of particles that are "attached" to it. Each leaf has its own 3D position and orientation relative to the tree's local coordinate system. The leaf particles are rendered with a simple polygon, mapped with an alpha-blended leaf texture like the ones in Figure 4.

For the orientation and size of the particles, I track the stroke until the user releases the mouse. The length of the stroke determines the scale of the rendered particle, and the direction of the stroke determines the rendering orientation. The orientation allows you to rotate the leaf to point in any direction, though the texture faces the camera initially. The leaves can either be rendered using the initial facing or be set to orient so they are always facing the camera. This creates a different effect that is useful in some cases.

When sizing the leaves with the brush strokes, some care must be taken. A bit of variation is necessary, but too much looks odd. Obviously, an extremely large leaf would look silly. I can always randomize the size a bit later if I want, but giving the artist some control is always good. I have also experimented with different types of textures such as other types of leaves, twigs, moss, and even mushrooms.

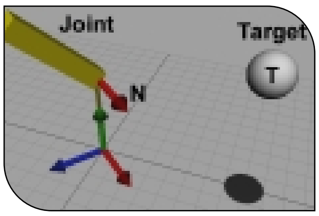
Different brush types and painting methods would create other effects. Attaching a spray nozzle and allowing the artist to "airbrush" leaves would give an effect similar to the Paint Effects system found in Maya. It's even possible to create things such as vines by rendering between two particles.

Because the leaves are treated as particles, they can be controlled just like a particle system. At the press of a button, all the leaves could fall off the tree and drift to the ground. By connecting the particles with springs, you could make vines that actually swing. I have just begun to explore the possibilities for this real-time dynamic, organic environment.

LISTING 1. OpenGL buffer to encode colors at the three vertices.
LISTING 2. Pseudocode for calculating the clicked point.

LISTING 1.
```
face = visual->index;
for (loop = 0; loop < visual->faceCnt; loop++,face++)
{
   glBegin(GL_TRIANGLES);
        glColor3f(1.0f, 0.0f, 0.0f); // Vertex 1 is red
        glVertex3fv(&visual->vertex[face->v[0]].x);
        glColor3f(0.0f, 1.0f, 0.0f); // Vertex 2 is green
        glVertex3fv(&visual->vertex[face->v[1]].x);
        glColor3f(0.0f, 0.0f, 1.0f); // Vertex 3 is blue
        glVertex3fv(&visual->vertex[face->v[2]].x);
   glEnd();
}
```

LISTING 2.
```
face = visual->index[triNumber];
v1 = &visual->vertex[face->v[0]].x;
v2 = &visual->vertex[face->v[1]].x;
v3 = &visual->vertex[face->v[2]].x;
click.x = (v1.x * vColor.r) + (v2.x * vColor.g) + (v3.x * vColor.b);
click.y = (v1.y * vColor.r) + (v2.y * vColor.g) + (v3.y * vColor.b);
click.z = (v1.z * vColor.r) + (v2.z * vColor.g) + (v3.z * vColor.b);
```



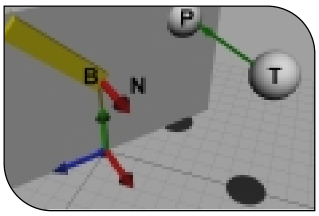


FIGURE 5 (left). The problem: Rotate a 1-DOF joint to a target.
FIGURE 6 (right). The approach: Project target on rotational plane.

The Trees Are Uprooting

My goal was to create a haunted tree, and in order to scare the children, the tree needs to be able to move. Because the tree is really just a single mesh object with a bunch of particles attached, I shouldn't have much problem. Just like a game character with a skeleton for animation, I can put a skeleton inside the tree and move it with matrix-deformation techniques ("Over My Dead, Polygonal Body," Graphic Content, October 1999). Procedural animation techniques can make the tree automatically sway as if the breeze were moving it around.

The only twist to the matrix-deformation system is the leaf particles. They need to deform with the rest of the tree, so my solution is to deform the particles using the same routine as the mesh. To do this, the particles need weights to relate them to the bone matrices. My simple solution is to have the particles adopt the same weights as the nearest vertex in the mesh. This seems to work well enough.

However, in order to give the tree the ability to reach out and try to grab the children, it needs to be able to reach a target. To accomplish this task, I am going to pull another tool out of my toolbox and use the inverse kinematics routines I developed a couple of years back. In order to use those routines for this application, however, I want to take a minute to flesh out some new IK ideas.

Inverse Kinematics Revisited

In this column back in 1998 ("Oh My God, They Inverted Kine!" September 1998), I described a system for iteratively computing the inverse kinematics for an articulated chain. The sample application was 2D, because it made the interface much easier (and I am lazy). However, the technique worked equally well in 3D, treating each joint as a three-degree-of-freedom ball-and-socket joint. To move toward the target, each joint rotated about an arbitrary axis to move toward the goal.

While quite effective at reaching the goal, treating every joint like a ball-and-socket led to some weird visual artifacts. For example, having a shoulder or hip joint behave like a ball-and-socket is fine, but an elbow or a knee shouldn't go twisting all around. At the time, I proposed using degree-of-freedom restrictions to combat this problem. Because I was using quaternions to represent the orientation of the objects, I had to then convert from quaternion to Euler angles while enforcing the restrictions and then convert back again. While this is not terribly efficient, it works well enough. There are times, though, when it doesn't make a lot of sense. A knee joint can be modeled efficiently using one degree of freedom. In order to implement this on my general IK system, I need to use degree-of-freedom restrictions to keep the other two angles equal to 0. This works, but it slows the convergence of the IK solution, because the restrictions are undoing work from the IK routine.

At the Game Developers Conference in 1999, I discussed my method of treating these 1-DOF joints as a special case. Instead of calculating the full 3D solution for these joints, I could project the target point into the plane of rotation for the joint, and optimize toward that goal. Since that time, I have received a bit of mail from people who attended that session and others who have run into this problem independently who were looking to understand the math behind this idea. This is exactly the type of problem I have attempted to address through this column by showing practical examples of how mathematics can be used as a tool to accomplish specific goals. Let me start by stating the problem.

The problem. I am given a joint in an arbitrary initial orientation that is free to rotate about one axis and a target position in 3D space. I want to find the angle to rotate the joint such that the end

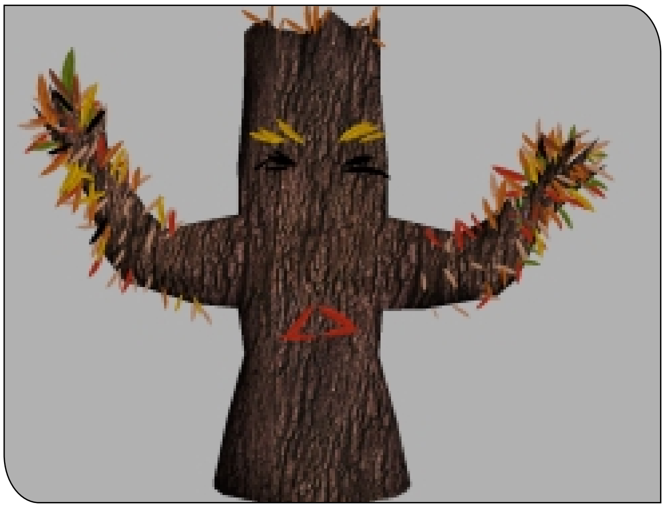


FIGURE 7. The final tree in action.

of the joint is as close as possible to the target.

For those of you who are good at vector math and geometry, this problem is an easy one. Now is your chance to exercise those math muscles. Set this magazine down, get a piece of paper, and work out the solution before reading the next section.

The approach. O.K., who cheated? I know that because the joint in Figure 5 is only allowed to rotate about axis *N*, the solution space is the plane where *N* is the normal to that plane. If I can find the nearest point to the target on that plane, that would be the new target point, *P*.

Start by creating a vector, *V*, from the base of the joint to the target. Take the dot product of this vector and the joint's axis of rotation, *N*. This result is multiplied by *N* to determine the distance along the axis, *D*. Subtract this distance from the target and you get the projected point, *P*.

$$\vec{V} = T - B$$
$$\vec{D} = (\vec{V} \bullet \vec{N})\vec{N}$$
$$\vec{P} = \vec{T} - \vec{D}$$

The dot product comes to the rescue once again. For those clever ones out there, we need to make a new *Schoolhouse Rock* cartoon starring the magnificent Product Brothers: short, stout Dot and his taller, nimbler brother Cross, fighting the evil force of Geometry.

This formula is useful for another purpose as well. The magnitude of the distance vector, *D*, is the closest distance from the target to the plane. That can be useful for applications such as collision detection. By using this new IK method, I can treat the tree's "elbow" joints as 2D joints, making the reach animation a bit more realistic. Check out the final tree in action in Figure 7.

Scary Monsters

Combining a 3D paint and particle system with matrix-deformation techniques has allowed me to create a Halloween tree with a bit of an attitude. If you move the "victim" close to the tree, it will start to shake and as the victim gets closer still, the tree will grab for him. Spooky, isn't it? Grab the demo and the source code off of the *Game Developer* web site at www.gdmag.com.

FOR MORE INFORMATION

WEB SITES

The best *Schoolhouse Rock* site on the Internet:
www.genxtvland.simplenet.com/SchoolHouseRock/index-hi.shtml

MOVIES

Disney's *Tarzan* (1999)

PUBLICATIONS

Daniels, Eric. "Deep Canvas in Disney's Tarzan." *Proceedings of SIGGRAPH 1999*, p. 200.

Meier, Barbara. "Painterly Rendering for Animation." *Proceedings of SIGGRAPH 1996*, pp. 477–484.

The Evolution of 3D Game Models — Part 2

The Characters of the Next Generation

For nearly ten years now, computer graphics in film have dazzled us with realistic simulations. As much as computer game artists wanted to create similar high-quality images, limitations of real-time graphics in videogames have left artists with imagined characters running around in their heads rather than on their computer screens. With the advent of powerful next-generation consoles, many of the chains that that have held back artists' creations can be broken, and a new era of expression will be possible.

But before we get carried away and compare the next-generation console graphics with the computer-generated graphics in the film industry, we must realize that game artists will continue to face limitations and difficult choices. First of all, most game development teams do not have the manpower nor the financing to add unlimited realism to their games. Second, the graphics for a movie only need to look good from one angle, on a 2D theater screen. In a game, the viewer has several viewpoint options, and the art needs to look good no matter which angle it is viewed from. Delivering high-quality graphics in true 3D space can take a lot of time, and a product cannot be in infinite production. There are always tight deadlines, and sometimes corners have to be cut. So, despite all the possibilities that will come with the next generation of characters, it is still very important to set priorities beforehand and figure out a way to get the most out of your graphics during your development time.

To break down where character improvements can be made most effectively, we should first look at the different parts that make up a character model. A character model is made up of 3D geometry textures that show color on the model, and animation or movement that brings the character to life. These three main groups are all the aspects that can be improved upon. A 3D model can be incredibly detailed, with everything from eye sockets to dilating pupils, and the artist has to decide where attention to detail will have the greatest impact. Better textures can make that eyeball look wet and shiny; the skin can have the impressions of the veins underneath. Animation is another area for character improvement. Animation can now be motion captured to get simulated movement — for instance, a character's mouth can be animated so that it appears to actually pronounce the words rather than "jaw" them. The artist will have to decide which of these aspects are the most important to make the characters come to life.

FINAL FANTASY. This level of texture graphics used in the upcoming Final Fantasy feature film will someday be possible for games.

Modeling

With faster processor speeds, the first place the artist will want to concentrate on is the model. The easiest thing to do would be simply to add more detail to the model. Rather than limiting the number of shapes because of tight polygon restrictions, artists now have a greater amount of extra polygons to work with, allowing for much greater detail. A head can now have a mouth, teeth, eye sockets with round eyes, ears, and whatever other details that fit within the polygon budget. To get the best results, it's a good idea first to add detail to the large shapes and make sure those are refined before moving on to details that might not be as noticeable. If the character is seen from a third-person point of view, it is best to spend more time on those polygons that will be the most visible. Of course, if the game has a lot of close-ups and character interactions, facial details immediately become very important. A downside of adding all this detail to a model is that the geometry can get really dense, and it is no longer possible to change a model quickly. Models can still be carefully built by hand, but some companies have already chosen to scan their characters instead.

For the game ODDWORLD: ABE'S ODDYSEE by Oddworld Inhabitants, the designers carefully built sculptures of their characters and then digitally scanned them into the computer. Although this seems like a long process, this might soon become a time-saver for building new models. For instance, Electronic Arts is scanning in the wrestlers for their latest wrestling game. Once a full-body scan of a wrestler is done by Cyberscan Technologies, game designers have a perfect replica. However, the dense geometry of this type of character can be very difficult to manipulate. Companies such as Paraform have created software specifically to make this data more

MAARTEN KRAAIJVANGER | *Maarten is the lead artist at Nihilistic Software, which just completed its first game, VAMPIRE: THE MASQUERADE — REDEMPTION. He is currently going crazy realizing just how much more time and effort it takes to make more lifelike game models.*

PIXAR SHORT

ABOVE. Pixar's Geri's Game featured subdivision surfaces.
RIGHT. A photorealistic wall from Insomnia Software.

manageable and able to be converted into different formats.

Until only a couple of years ago, most programs only supported modeling methods with either polygons or curves. When Pixar created their Oscar-winning short film *Geri's Game*, their resident Ph.D.s came up with a new modeling method: their models were built with patches called subdivision surfaces. The incredible benefits of subdivision models are that the geometry is less dense and can be manipulated fairly easily. In addition, the artist has the option to put more detail where it is needed without having to apply it throughout the model. With limited RAM on consoles, the smaller footprint of a subdivision model could make this format very attractive to a designer, because many detailed polygonal characters chew up RAM very quickly. If the subdivision surface model can be used as a simple yet detailed template of the polygonal game model, it could very well become the standard in the next generation of character development.

Textures

Another improvement for the next generation of character models will be in the textures. With the introduction of texture compression, a lot more can be done with the texturing of models. Effects such as bump maps that previously could only be done in movies are going to become commonplace within games. New technologies have been introduced that have made creating photorealistic textures easier. When a 3D model or a life-size person is scanned in three dimensions, the color on the model can be included with the scan, making it possible to get the exact color of every hair, pimple, and mole, with the exception of eye color, because the eyes must remain closed during scanning. With the introduction of digital cameras, it is now easy to get perfect textures of every wall and billboard, and achieve all the gritty detail without the hassle of developing film. And finally, the tools to paint the textures on the computer have developed further to make it even easier to paint them in 3D space rather then in 2D.

On the Playstation, the textures that made up the models were 32¥32 and had 256 colors; on the Xbox, the textures can be as big as the programmers let you make them. Best of all, the Xbox can do multiple rendering passes on the textures. A brick wall that used to be composed of a flat-looking color map will now feature a bump map to push the mortar back a little; a spectacular texture map can be created to make the wall appear to those walking by as if rain has been dripping down the side.

With all the possible texture effects becoming available to make the game world more realistic, the availability and ease of digital photography has made the color aspect of textures much more lifelike. If we are striving to capture reality, what better way to create a texture than to take of picture of it? Some games in development have already taken the plunge and created their entire environments from digital pictures. MAX PAYNE by Remedy Software is one of the first games to rely heavily on photographic imagery for their characters and world. The results are a unique look that shows off all the detail of what you might find in the gritty city. These photorealistic environments will become more common because it is much cheaper and less time-consuming to take a picture than for an artist to paint the texture from scratch. Of course, photorealistic environments are not suited to every game, but because it's easier to create higher-resolution textures and create a more realistic environment, it is a good fit for many next-generation games.

However, there can be some problems using digital photography in games. Because most games attempt to create a world that blends all of its elements seamlessly, mixing hand-painted textures with digital photography can look jarring. Because the detail of digital photography is a perfect representation of a texture, the hand-painted textures created by the artist might stand out and break the illusion. One common solution is to use a digital photograph as a template and paint a texture from it. This can reduce the realism and give the world a more unified and unique look. But if the art team is only using unenhanced digital imagery, they might find themselves stuck with having to do all the other textures in the game with digital photography.

Animation

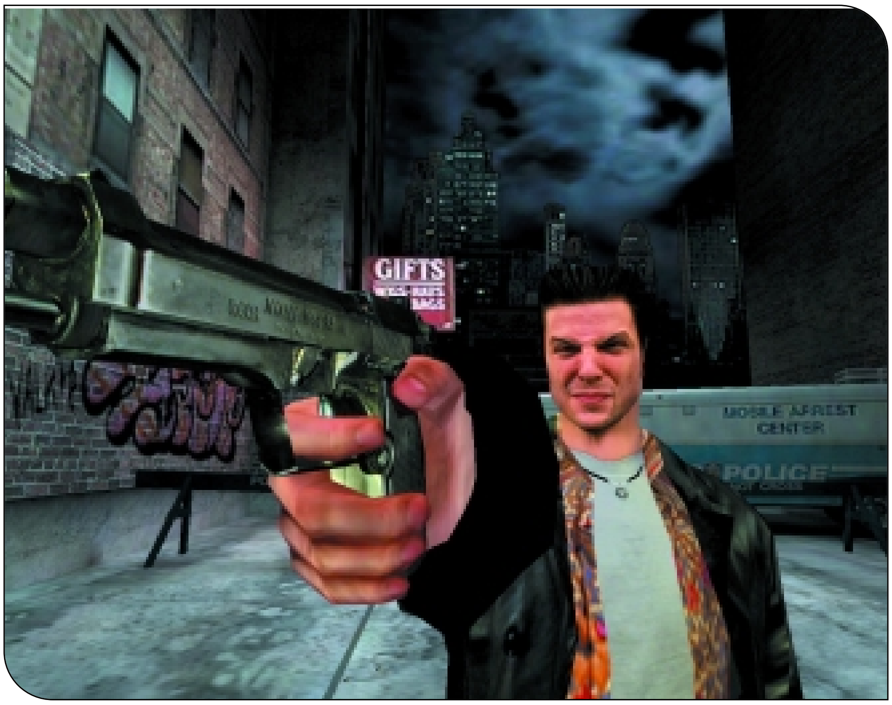
One of the biggest advances in the next generation of game models will be in animation systems. Most of the animation systems in games so far have either relied on motion capture or on traditional keyframe animation. So far these systems have been great, but they have their limitations.

Motion capture has been used in a lot of sports and fighting games, but it is not appropriate for every game. Motion capture does exactly what rotoscoping did for traditional 2D animation: it is capable of capturing fluent human movement and the results are extremely realistic. From timing to the weight and mass of a character, motion capture provides human simulation to perfection.

The problem is that in most videogames you don't want perfect movement, but rather something more dynamic and flashy. You wouldn't want an average Joe for your character design, you want your character to be larger than life. The same is true for how the character animates. When doing motion capture, you can hire an actor that is aware of his movement and gives the motions more "oomph," but no matter what the actor does, he is always constrained by the laws of gravity and physics. The actor will always move and behave in a recognizably human manner. An animator creating the motion by hand, however, does not have the same restrictions, and is free to animate the character with creativity regardless of what reality might dictate.

The downside to this is that it takes much longer to do each animation by hand, and the costs can add up quickly. Because it can be very expensive to do traditional animation, many companies prefer to do motion capture. For many games it can work very well, but even when it is pulled off perfectly it has many limitations. Because motion capture can, for the most part, only be done by and for humans, what do you do for all the other nonhuman critters running around in your game? You can tweak the motion capture data to fit different humanoid skeletons, but what do you do about crazy fantasy creatures? Monsters can have eight limbs, two heads, and five arms, and there is no way to capture the motions such creatures would need to be brought to life. If the developer wants to use motion capture for the humanoid characters and hand-animate the creature, the game developer is once again faced with the dilemma of mixing media, much like mixing digital photorealistic textures and hand-painted textures creates a disjointed feel. It's possible if the animator for the monsters is very talented, but it would be very difficult to match hand-animated movements to smooth, motion-captured animations.

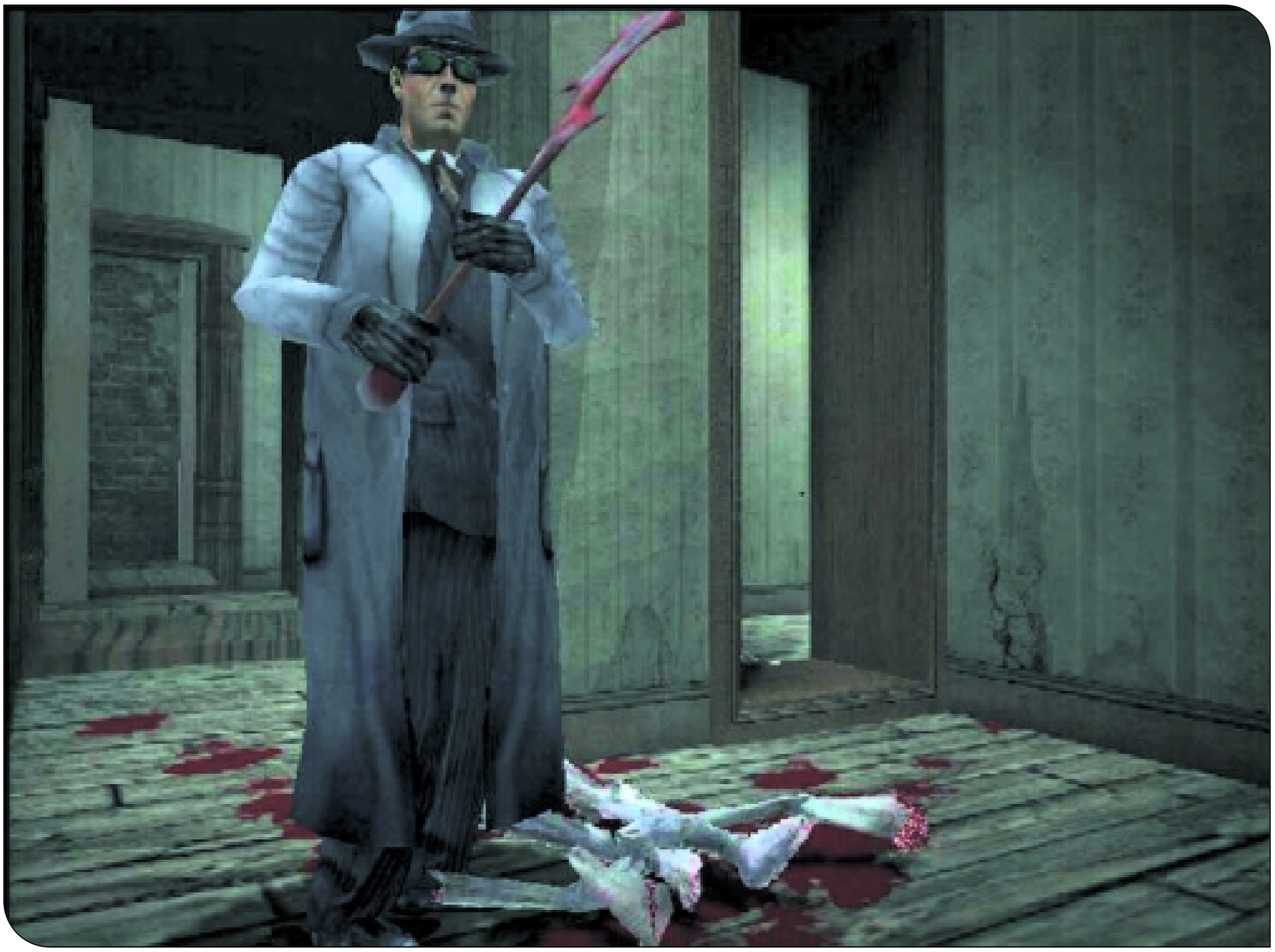
For the next generation of games, it will not be an easy solution to pick between motion capture and traditional animation. Because we can never motion-capture all the creatures of our imagination, traditional computer animation skills will always be in high demand. What can be done is to examine traditional animation and figure out ways to make it a faster and more efficient process. This is where real-time dynamics comes into play.



MAX PAYNE. It is one of the first games to rely heavily on photographic imagery for textures.

Real-time dynamics is where a skeleton or 3D geometry reacts according to the laws of physics. The use of dynamics in computer graphics has been around for a long time and has been used extensively in film. Convincing cloth dynamics, on the other hand, have only been achieved within the last three years. *Geri's Game* was the first to show cloth behaving as one would expect. The clothes folded and creased as you would expect them to. Your eye was no longer paying attention to the effects, but rather to the character itself.

So far, real-time dynamics' use in games has very limited ways in games because it squeezes the pulp out of any supercomputer. In NOCTURNE, developers at Terminal Reality took the plunge and attempted to do fully dynamic clothing. The results were a mixed bag. The fact that the clothes on characters moved and animated differently depending on the situation added quite a lot of realism, but the game ended up having high system requirements and the clothes did not have much variety in the way they moved. To do clothes correctly, you need to have control over properties such as fabric, elasticity, and weight. For example, a piece



NOCTURNE. The fact that the characters' clothes animated differently depending on the situation added a lot of realism, but came at a performance cost.

of corduroy should not move the same way as a piece of silk. Many of the features that provided the realistic cloth simulation in *Geri's Game* have been incorporated into advanced tools such as Maya. Because it is now easier to incorporate cloth in 3D animation packages, it is just a matter of time before such textures are used successfully in character models.

Another aspect of character animation that is on the horizon is full 3D lip synch for in-game characters. So far, the lip synch in most games consists of the mouth simply moving up and down without any mouth shapes. Lip synch with actual pronunciation of words has been done in some games with the use of textures, but not with the mouth actually deforming to say the words. Facial animation setup can be quite complex. Although tools to create 3D lip synch are available in most next-generation packages, most of them are not very efficient for real-time games. The most popular method requires the artist to make multiple copies of the same head and create different expressions for each one, which are usually referred to as morph targets. The lip synch is then accomplished by actually blending the geometry between all the different copies with all the different heads. Whichever method is chosen, they all present long setup times and lots of programming to work properly.

Conclusion

With all the new technology cropping up, it is an exciting time to be a game artist. With fewer limits, game artists will be able to bring exciting new characters to life. Now more then ever, game artists can pursue many of the techniques and methods pioneered by the computer graphic artists working in film. Artists need to make sure they are familiar with the techniques and tools used so they are capable of making well-thought-out decisions. All the power unleashed with the new consoles and graphics cards have made the artistic possibilities virtually endless, yet the resources to create them have remained finite. With the ability to do so much more, artists must use discretion and focus on features that will make the biggest impact to separate them graphically from their pack of competitors. The starting gun for the next generation of games has fired, and I can't wait to see what's going to be leaping off of the screen.

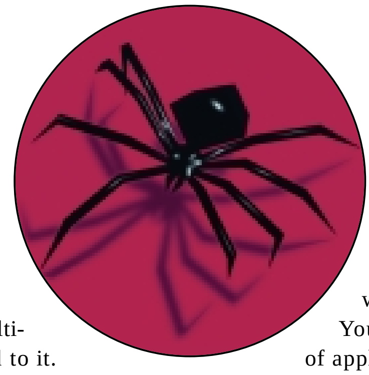
Discuss this article in Gamasutra's Connection!
www.gamasutra.com/discuss/gdmag

Return

mark peasley

Arachnophobia

Animating a multi-legged creature



So here's the problem. I'm minding my own business, when along came a game designer with this great idea of making a Spider Dungeon in our fantasy role-playing game. If you're like me, you always thought about doing that multi-legged insect animation, but never got around to it. Well, here's your chance.

In this article, I'll cover the basics of how to set up and animate an eight-legged spider. I'll also discuss how to create a simplified sequence of leg/foot placements that can be looped.

The target output for this animation is a run-time 3D environment. What we end up with is a single looping animation which the code will take and translate the necessary distance, causing the spider to appear to walk on the terrain. Because of the limiting factors in run-time development, we need to create a compact yet believable animation, in as little time as possible.

The software I used for this tutorial was 3D Studio Max 3 and Character Studio's Physique modifier, but the general processes should be applicable to most of the 3D packages out there. I suggest you have a fresh supply of java (the beverage, not the code) readily available to help you get things off on the right foot.

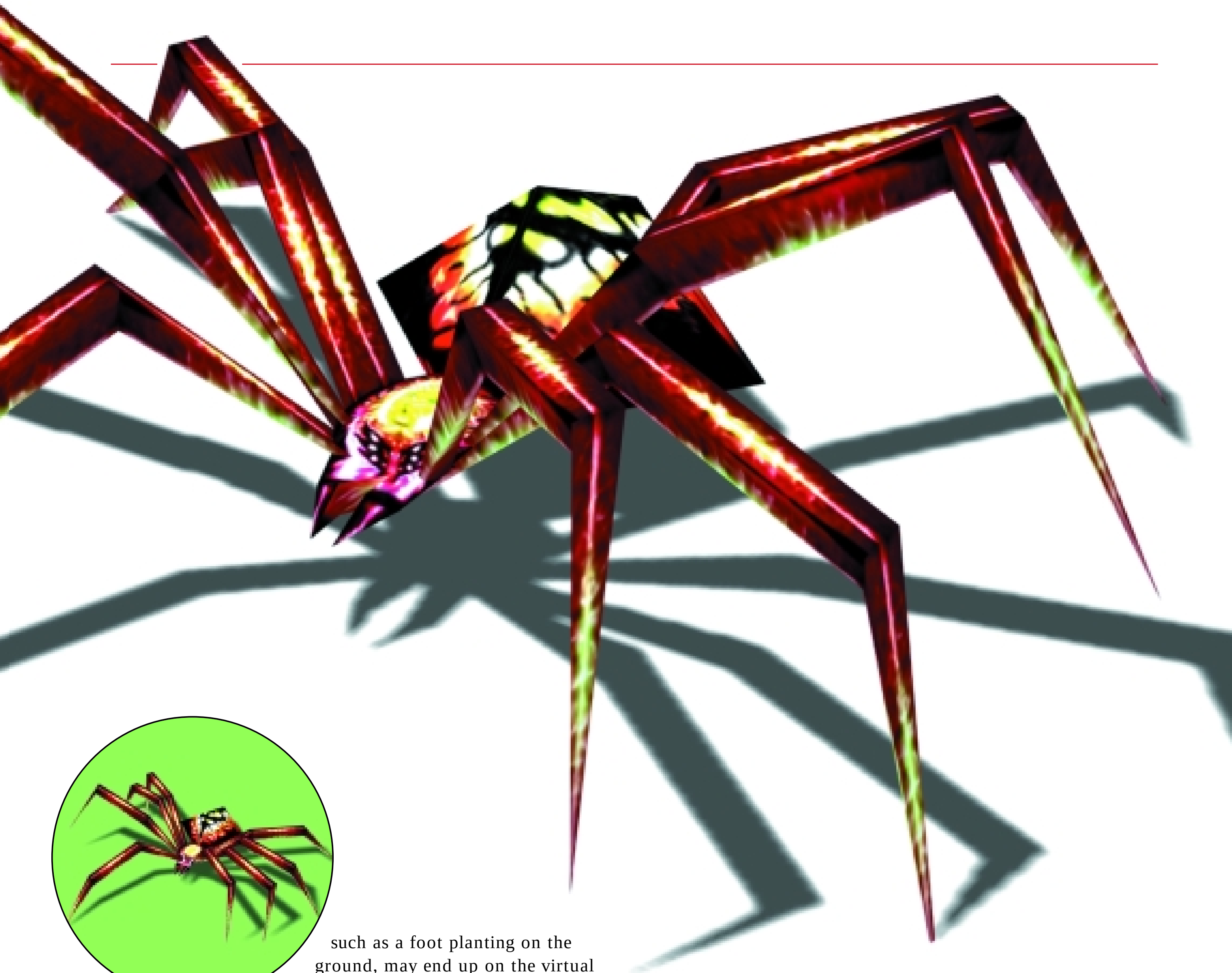
References: An Important First Step

Oftentimes, even the most seasoned animator will want to bypass mundane research and cut to the chase. It's even more tempting when the subject is something familiar. But therein lies the danger: If the subject matter is familiar, then your audience will have a much more discriminating eye for errors or anomalies. By studying a few reference video sequences or multimedia clips, you can pick out subtleties that you will need in order to bring your animation to life. Your local library or video store holds a wide variety of applicable references.

Establishing the Frame Rate and Step Sequence

The frame rate (in frames per second, or FPS) you establish for animating is very much dictated by the type of game you are working on and the speed at which you want the subject to travel. In this particular example, the animations will be exported and end up controlling a single skin mesh with a skeletal structure in a run-time 3D environment. While a high frame rate is expected, we intentionally animate at a much lower frame rate. Our target rate will be 12FPS.

There are several reasons for choosing this lower frame rate. If the animations are done at, say, 30FPS, and the game ended up going below that speed during gameplay, then the code would need to truncate some of the keyframes. In most cases, this means sampling the data at a lower rate. Now the problem becomes determining which keyframes are more important than others. Most of the time, resampling simply removes every other keyframe or every third one. This can lead to disaster in the final animation, because some of your more important keyframes,



such as a foot planting on the ground, may end up on the virtual cutting room floor. If critical keyframes are removed because of the data sampling, the entire animation starts to fall apart.

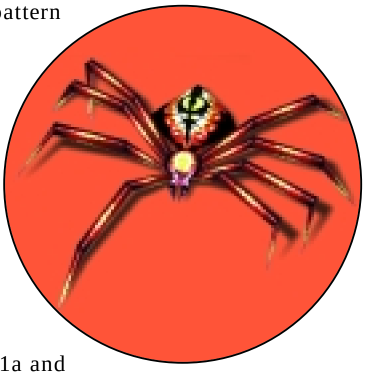
Thus, animating at a lower frame rate avoids this problem. Initially, this may make it more difficult to get the results you are looking for, but it also forces you to become more aware of which motions and poses are most critical to conveying the movement.

The downside to having fewer keyframes is a less subtle animation. Often, secondary motion is difficult to capture, and having to loop the animations eliminates some of the follow-through motion as well. If the game engine is robust enough, blending and spline interpolation can compensate for some of the lost motion. With some work and planning, a 12FPS animation need not end up looking like Saturday morning cartoons.

The next task is establishing a looping foot sequence. Just as in traditional cel animation, the two extremes of the leg position as well as the maximum leg-height frame (also known as the passing frame in a human walk cycle) need to be established. In the case of the spider walk loop, I indicate a local Z-axis frame height with an arrow. This frame is where the leg, as it moves through the air, reaches its maximum height.

After studying the walk loop of several spiders, I've determined a cycle that looks good enough to use in a run-time environment and is fairly compact. This cycle isn't exactly like a real spider walk sequence, but it approximates it close enough for our purposes and has a fairly low keyframe count.

After some close study, I discovered that each leg on the alternate sides of the spider's body is doing essentially the same thing. The only difference is that the sequence of when they plant and when they transition are simply out of phase with one another. This repetitive pattern is echoed for each set of legs from front to back. Once I have the whole sequence, I then determine a loop point.



The length of the sequence has a direct bearing on how fast the spider walks. In our case, we are animating at 12FPS, and have decided that our in-game spider should move fairly fast. After some experimentation, I decided on a sequence of 12 frames (see Figures 1a and

MARK PEASLEY | *Mark has been in the game industry since the late 1980s and is currently the art director at Gas Powered Games. When he's not crackin' the art whip, he referees three boys at home and enjoys scuba diving in the Puget Sound. Visit his web site at www.pixelman.com or e-mail him at mp@pixelman.com.*

1b). This means that one entire looping cycle occurs every second. This is much faster than a tarantula-type walk loop, but it is intentional because our spiders need to be able to chase down characters within the game.

The Basics

I will assume at this point that you have already modeled and texture-mapped a mesh. In the case of our spider, we have a fairly low polygon count that weighs in at around 175 faces. The polygon count is low because we ultimately want to have several dozen spiders on-screen at once. In addition, our game camera is far enough away during the scenes in which this type of spider is present that we can make very simplified leg joints. The flattening that occurs at these joints happens quickly enough that it isn't really identifiable. (Later on in the game, we created a higher-polygon spider that can stand up to much closer camera views without breaking down.)

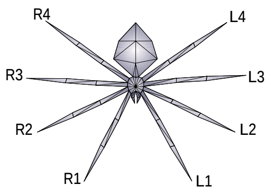
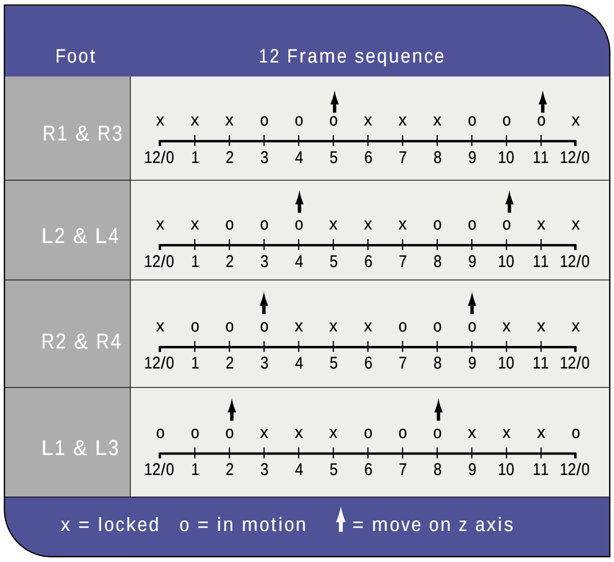


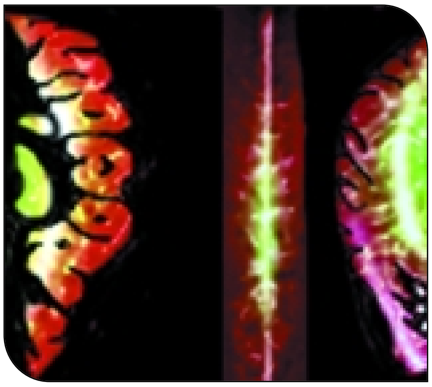
FIGURE 1A (above left). The references to the spider's legs used in this article.



Foot	12 Frame sequence												
	12/0	1	2	3	4	5	6	7	8	9	10	11	12/0
R1 & R3	x	x	x	o	o	↑ o	x	x	x	o	o	↑ o	x
L2 & L4	x	x	o	o	↑ o	x	x	x	o	o	↑ o	x	x
R2 & R4	x	o	o	↑ o	x	x	x	o	o	↑ o	x	x	x
L1 & L3	o	o	↑ o	x	x	x	o	o	↑ o	x	x	x	o

x = locked o = in motion ↑ = move on z axis

FIGURE 1B (right). Each foot is locked to the ground plane for three frames, then moving for three frames. On the third frame of each moving sequence, the foot is raised on the local Z-axis.



While not critical, you can save yourself some time later on by making each leg an identical copy of the other, all radiating the same distance from a center point in the body. This will make placing and aligning the skeletal bones easier.

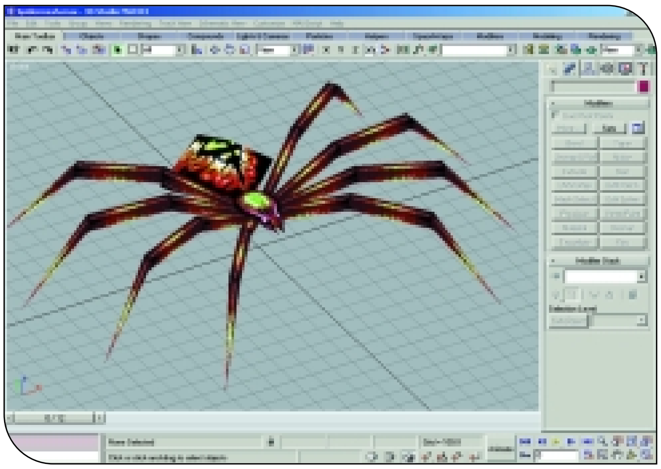
Figures 2a–c show an example of the spider as well as the texture map used. The texture map is 256¥256¥32-bit. Because of graphics hardware requirements within our particular game, the maximum size we can go to is 256¥256.

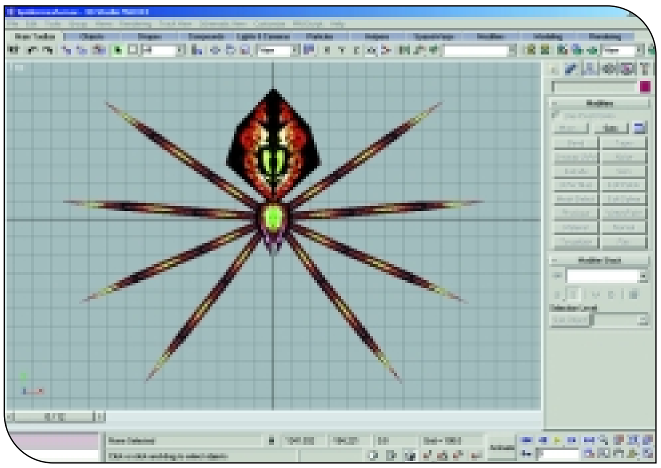
When considering what size to make your initial texture map, bear in mind your final camera distance. Often, modelers and animators work in full-screen mode in their paint and animation software, and it's easy to forget that your character may end up the size of a dime on the game screen, with all the extra effort put into the texture lost. This is especially true if the texture is being MIP-mapped down. After some testing, we found that our spider texture was using its third-level MIP-map (64¥64) at the default camera view.

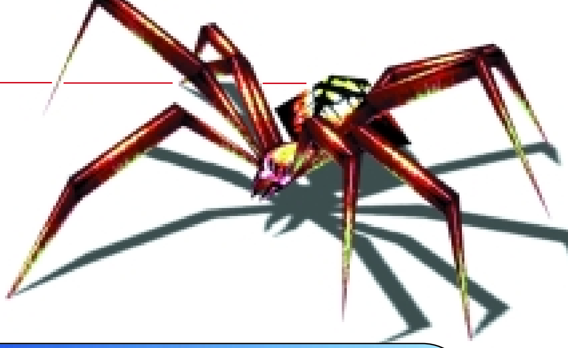
Creating the Skeleton

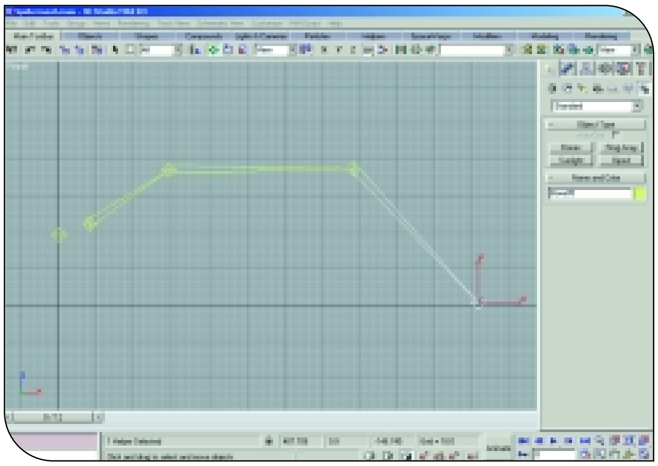
For this animation, we will create a combination inverse and forward

FIGURE 2A (left). The texture mapped to our spider model.
FIGURES 2B (below left) & 2C (below right). Two views of our spider model, ready for animating.









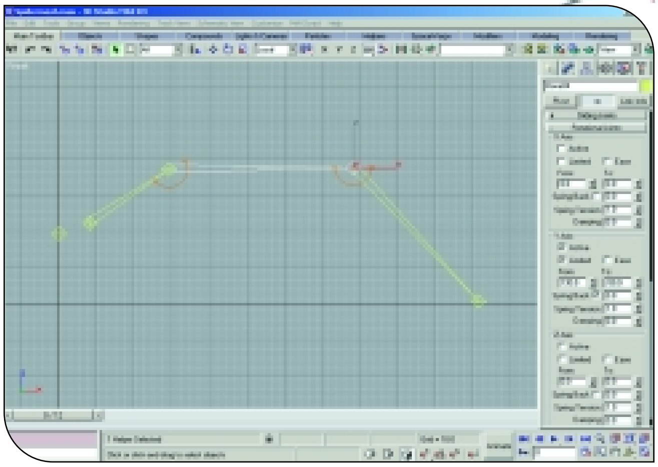
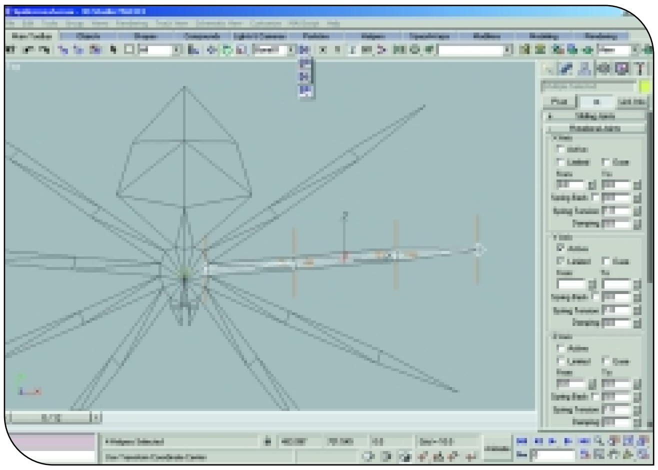


FIGURE 3A (above left). Creating the bone in a standard view.
FIGURE 3B (above right). Setting the constraints on the leg bones.
FIGURE 4 (right). Rotating the bones into position.



kinematics skeleton. This allows the best of both worlds in terms of IK foot control and FK body control. The IK skeleton legs allow much easier control of the foot placement and subsequent editing, while FK bones will control the body and "fangs" of the spider. The IK legs also allow us to animate the body of the spider after the leg sequence has been created without altering or affecting the foot placement. With an FK-only system, subtle adjustments to the body at the end of the animation process would alter all of the leg positions, requiring a fair amount of reworking.

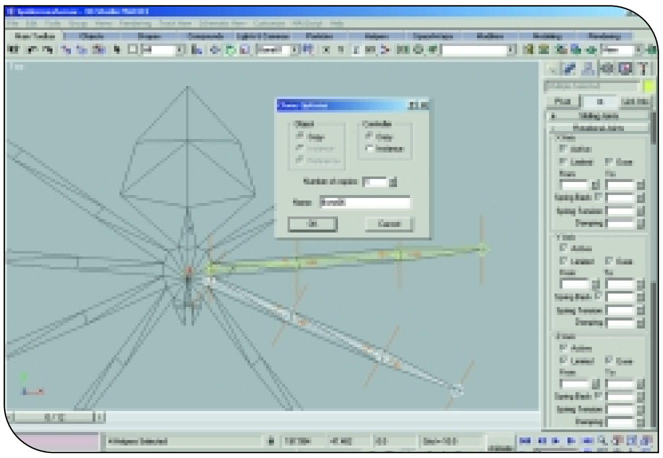
The first step will be to create one leg, with all of the correct constraints, which we can then clone and rotate. The resultant array of legs will have all of their constraint settings automatically duplicated. This saves some time by eliminating the need to re-input the settings for each leg bone separately. It also saves us from trying to set constraint limits on bones that aren't perpendicular to one of the main axes. We build the first leg bone perpendicular to an axis, set the constraints, then rotate it into position.

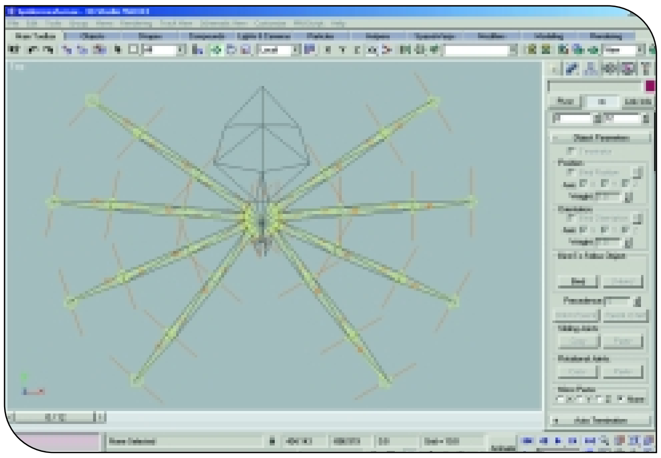
The root bone. In Max, the initial root bone is created as an FK bone. This isn't a requirement, but it makes animating the basic body motion possible without having to add a linked dummy to the scene. It also terminates each of the IK leg chains. Once we create the initial root bone, each of the leg bones will be generated, constrained, and linked to the root bone. We will place the initial root bone in the center of the body, at the point where the legs converge.

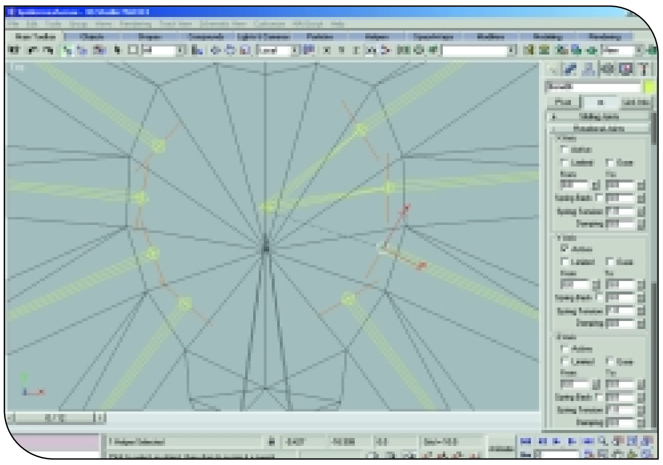
Setting up a single IK skeleton leg. I'll create a single leg bone in one of the standard left/front/top viewports. In Max, this facilitates the constraining process. By creating the bone in a standard view, the constraints will fall on either the X-, Y-, or Z-axis. If the bone was created from a "user" viewport, the process of setting the joint limits would get much more difficult to lock down and get predictable results. Once we have created and constrained the bone, it can be rotated into the desired position. (See Figures 3a and 3b.)

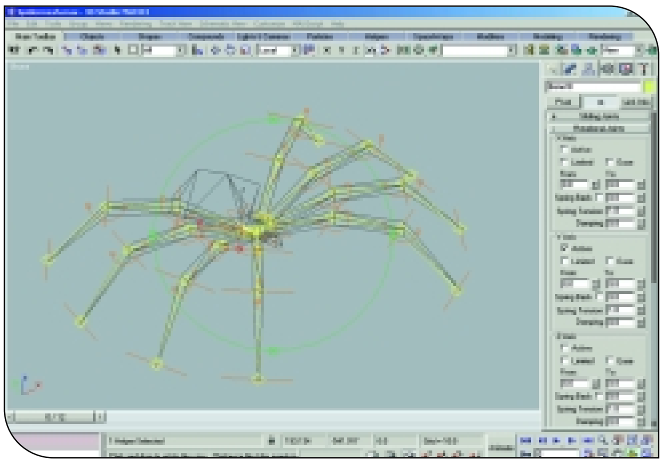
Making the bones behave. In Max, the IK bones are continually trying to solve themselves. One downside to this is that the default settings are a fairly low granularity. This tends to make the bones "settle" over the course of a few frames once the end effector is added to the IK chain. Even if there are no keyframes, the skeleton will visibly shift over the first five to ten frames. To avoid this settling effect, we need to crank up the IK Control parameters. We set the thresholds for both the position and rotation to 0, and set the IK solution calculator to a high number, such as 500. The end result is that the bones will lock into position with no visible settling effect.

Setting the constraints. The trick to setting the constraints is to do as little work as possible. The ideal leg allows the necessary control with as few dummy objects as possible. In this example, because the leg of the spider is mechanically very simple, I can use a single foot dummy. More dummies can be added to control each joint, but this additional control comes at a cost. As you begin to readjust the legs, each of the additional dummies will need to be adjusted as well. In this particular example, a single dummy was adequate to control the leg movement. While setting the constraints, you should continually test the motion extents of the limb. It is possible to readjust these after the animation is well under way, but this often creates undesirable side effects that may









FIGURES 5A (top left) & 5B (top right). Cloning the first leg and placing the clones. FIGURES 6A (bottom left) & 6B (bottom right). Making adjustments to individual legs and linking them back to the root bone.

require you to rework some of the animations. (See Figure 4.)

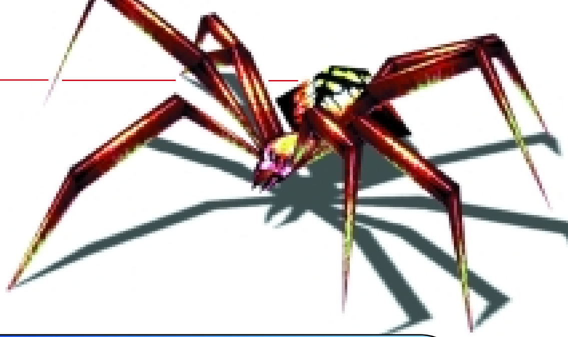
Fitting the bone into the mesh. The next step is to fit the bones inside the leg mesh so that when we clone the leg IK chain, it will fit into the remaining mesh with minimum adjustment. This assumes that you created your mesh in a semi-orderly fashion and that the legs are identical copies, rotated into position. If you didn't create the mesh this way, then you can easily fit each leg bone into the mesh manually, it just takes a bit longer.

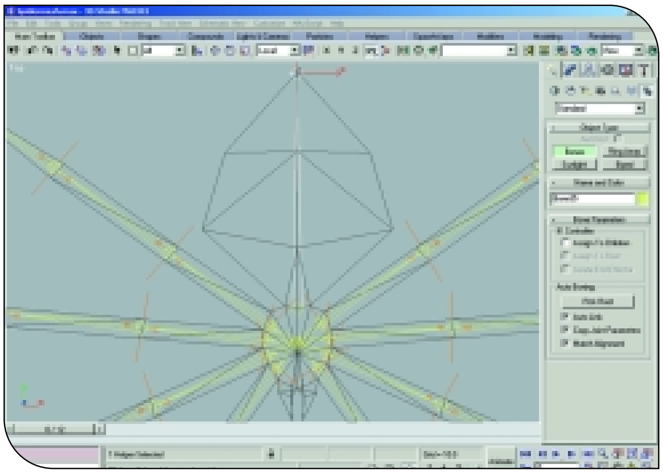
Making multiple legs. Now that we've set the constraints and parameters correctly for one leg, any cloned duplicates will bring along the settings that were made for the first leg. This saves a lot of repetitive input. The initial leg is selected, copied, and rotated into each new leg position. Because the initial root bone is in the center, we can use it as the rotation point for each duplicate leg. This places each clone of the leg IK chain in an appropriate location with a minimum amount of correction needed. (See Figures 5a and 5b.) We can make any minor adjustments to the individual legs and then link each leg back up to the root bone. (See Figures 6a and 6b.)

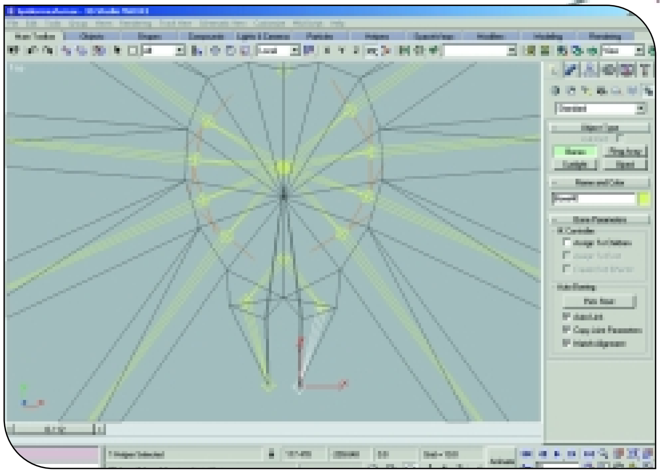
Creating the rest of the skeleton. For the rest of the spider, I used FK bones. This allows me to keyframe the body and the mandible movements without having to add IK end effectors and corresponding dummy objects to control them. Once again, I recommend using a standard viewport so that any constraints you may want to place on the bone can be done so with minimum effort. (See Figures 7a–c.)

Attaching the mesh to the bones. The next step is to attach the mesh vertices to the skeleton system. In Max, this is done by using the Skin modifier, or a Physique modifier if the Character Studio plug-in is installed. I prefer the Physique modifier because it allows type-in weights on individual vertices. It's much easier to get precise results using the type-in weighting method rather than the envelopes available in the Skin modifier. Also, the Skin modifier has some anomalies where bones can't be renamed after the modifier has been applied, so reworking is painful.

Speaking of naming bones, save yourself some trouble and give them logical names. It is much easier to skin or physique a skeleton that has the bones named something other than BONE02, BONE03, and so on. You may find yourself jumping in and out of the weighting menus in order to identify bones correctly if no initial effort was made to give them logical names.







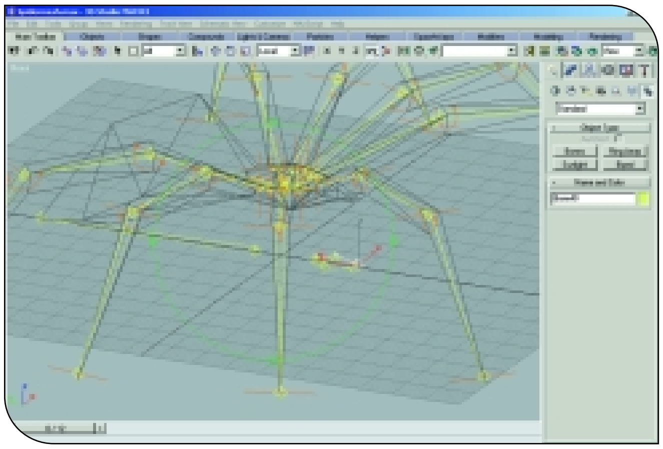


FIGURE 7A (above left). Building the spider's body with FK bones.
FIGURE 7B (above right). Aligning the mandibles to the mesh.
FIGURE 7C (left). The user view allows for accurate bone placement.

I could write an entire tutorial just on vertex weighting techniques. Because this article deals with animating more than skinning, I'll just wave my hands and your mesh will magically be ready for animation. This should be familiar ground for most of you, so I'll assume that you have skinned or physiqued your model before moving on to the next step.

Animating the Spider

First, we will create a couple of dummies to control the root bone and one front foot. Eventually, each of the spider's feet will have its own dummy, but it is generally easier to create and animate one foot at a time. The first dummy is linked to the root bone and used to control the body translation and rotation. While this isn't required, I find it easier to have a dummy for the root. The foot dummy is used to control the IK chain of one leg. Here, I start with the R1 foot. Once the dummy is in place, I place an end effector on the last leg bone. I then link the end effector to the foot dummy and it is ready to animate. (See Figures 8a–c.)

Simplifying the view. When animating, it's a good idea to eliminate all of the on-screen noise that has a tendency to creep in. By noise, I'm referring to all of the bones, constraint angles, end-effector targets, and other things that your animation program displays by default. In Max, I hide all of the bones I'm not directly using. I also "freeze" the mesh, which makes it visible but not selectable. In addition, I either hide or freeze all of the dummies except for the one I am animating. An added advantage of this is that hiding tends to speed up the screen redraw. On a simple character like this, it generally isn't a problem, but on more complex ones, it can have a big impact.

Moving the spider. You should now have only a spider mesh and two dummy objects visible, with everything else either hidden or frozen. The next step is to determine how far we should move the spider body as it is walking. This is dependent on the stride length, the number of strides a single leg makes in the animation, and how you want it to look. By sliding the foot dummy along its local Y-axis, you will begin to see the extents to which the leg can travel comfortably without clipping into other legs. Look at it in various views and then measure the distance. If you take the stride distance a leg covers and multiply it by the number of strides that the leg takes during the entire animation, and then multiply the result by two, you will find distance the body will need to travel during the course of the animation. In our case, this comes out to:

stride distance (600) ¥ number of strides (2) ¥ 2 = body travel distance (2,400)

This number gets you into the ballpark, but you will probably want to adjust it to suit your needs. Once you have determined a travel distance, you then animate the root bone across that distance via its linked dummy. In the case of this spider, it was translated 2,400 units on the local Y-axis. I made a keyframe for Frame 0 and again at Frame 12.

Animating the right front leg. Once you begin animating, it will become painfully obvious that the leg and its dummy stay in place while the rest of the spider moves. This is because as soon as you create a position end effector for an IK chain, it starts to solve for that target.

This is one of the primary reasons for not making dummies and end effectors for all of the legs at once, because they would all target their respective dummies at Frame 0. Several work-arounds can solve this problem, but I find it easier just to do them one at a time.

Next, we'll animate the foot to begin our walk loop. Now is a good time to review the original step sequence (Figure 1b) to familiarize yourself with what the R1 foot will be doing. Because the foot is "planted," or locked to the ground plane, during the first three frames, we need to make the first frame the fully extended position of the leg. Once it is keyed at Frame 0, the dummy will remain in the same place as the body of the spider continues to move. This causes the leg to compress as the distance between the dummy and the body decreases. If you've made any errors in your constraint angles, they will begin to show up now, during the extension and compression of the leg.

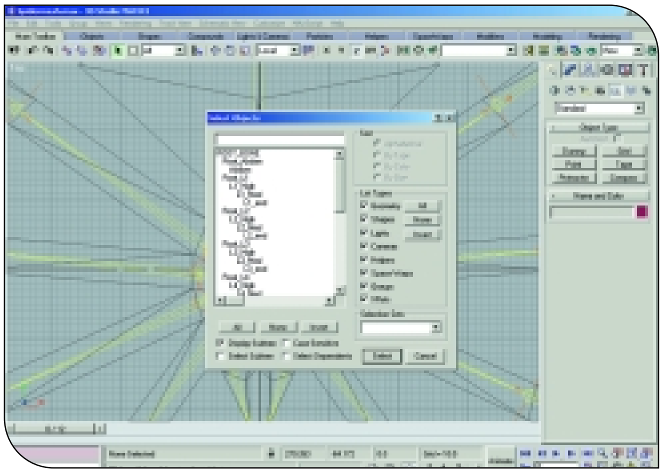
At Frame 3, the dummy is lifted. It makes a full extension once again on Frame 6, where it is keyed. The dummy is now keyed in position and remains there until Frame 8, where another key is set. On Frame 9, it is again lifted in the air and then plants to the ground plane again on Frame 12. Frame 12 should be an identical copy of the leg pose at Frame 0, because this is the transition frame for the looping cycle. (See Figures 9a and 9b.)

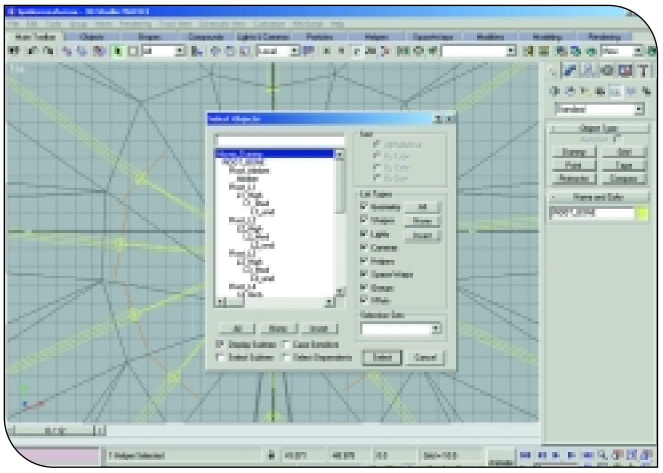
Each time the leg extends (Frames 0, 6, and 12), it is essentially the same pose. The only difference is that the body has translated or moved. One down-and-dirty way to get the same "pose" for each leg extension is to go to Frame 0, and duplicate or snapshot the mesh. This creates a mesh that has no bones but is frozen in the same pose as the mesh being controlled by the skeleton.

Once you have a mesh snapshot, you can move it to the new position on Frame 6. With the snapshot in place, it's easy to adjust the dummy so that the leg position is nearly identical to that of the original one. This is critical to do on Frame 12, as any variance in the leg pose will be seen as a pop in the animation. Once the duplicate mesh has served its purpose, delete it. After the entire leg sequence is done, play the animation and admire your handiwork. Viewing the animation at one-half and one-quarter speed can often highlight problem areas. Try to focus on getting the one leg correct before moving on to other legs. (See Figures 10a and 10b and Figure 11.)

Animating the left front leg. Now create a dummy for the left front leg (L1) following the same procedure used on the R1 leg. Keep in mind that each leg you animate has a counterpart on the opposite side of the spider's body that is animating exactly the same way — the two are simply out of phase with one another. So, each time the right leg is fully extended, the left leg should be fully compressed, and vice versa. Using the snapshot method of duplicating the mesh in a given position and then mirroring it makes the process of establishing extension positions for the opposite side fairly easy.

Because frames 0/12 of the L1 leg involve a "mid-stride" key, it's easier to animate the middle-leg moves first. After doing these, we can use the leg positions as a reference for the more difficult transition frame.





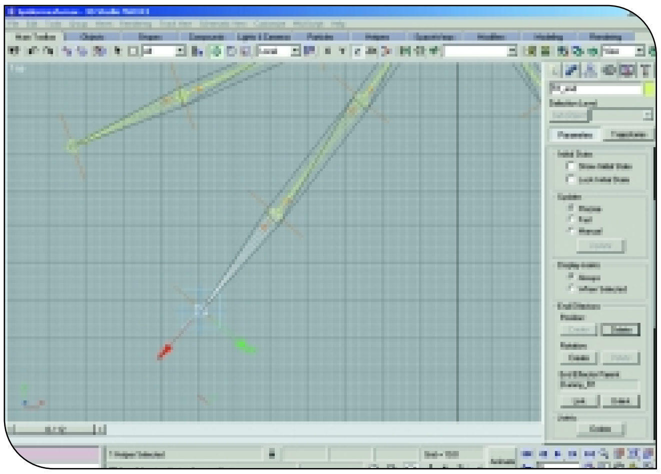
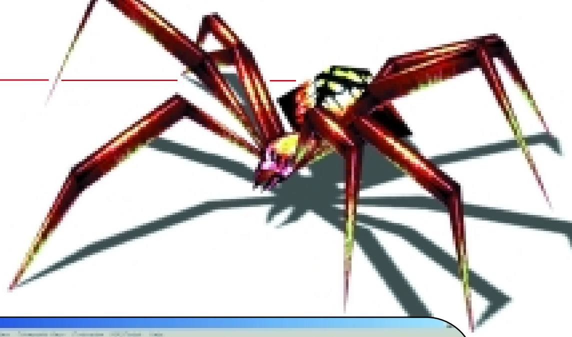
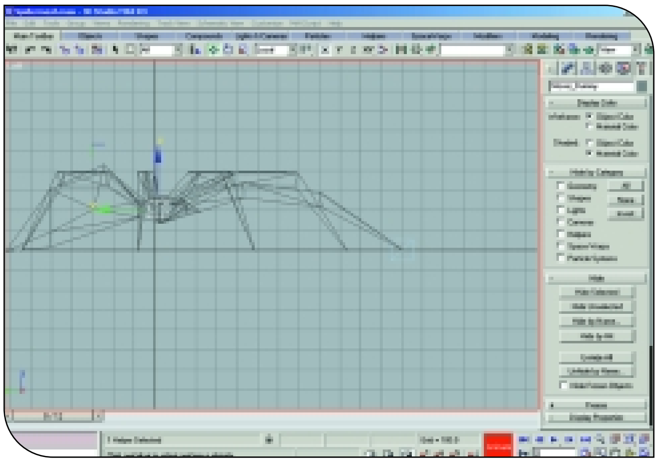
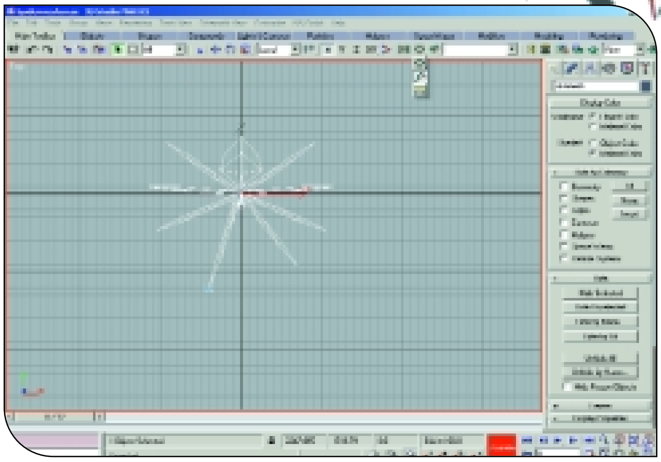
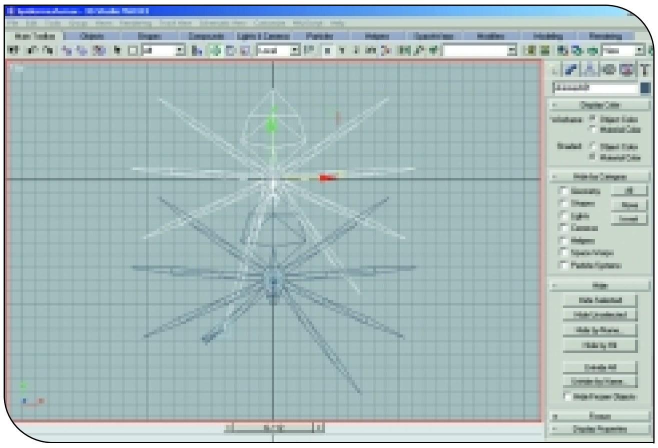


FIGURE 8A (left). Verifying the correct bone hierarchy.
FIGURE 8B (below left). Linking the dummy to the root bone.
FIGURE 8C (below right). Placing an end effector on the last leg bone.









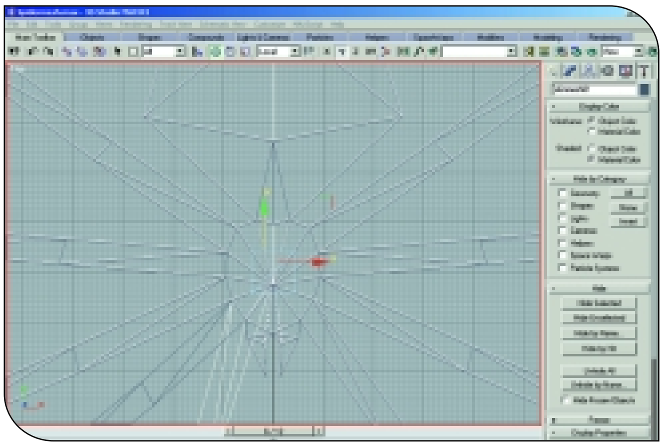
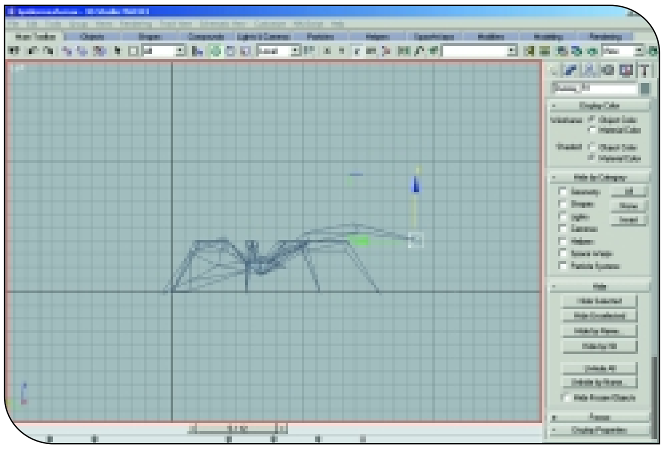


FIGURE 9A (top left). Front leg in fully extended position. FIGURE 9B (top right). Creating a duplicate snapshot mesh. FIGURE 10A (middle left). Using the snapshot mesh for reference. FIGURE 10B (middle right). Aligning the snapshot mesh over character mesh for keyframing. FIGURE 11 (right). Keyframing the vertical foot movement.



Go to the first frame where the left leg is planted and fully extended (Frame 3) and begin the process of animating the leg. Continue all the way through until Frame 11 is keyed and done.

The final thing we need to do for the left front leg is to create the first and last keys for the transition. On the right leg it was easy, because we started the motion after the Frame 12/0 loop point. For the left leg, we need to create keys for the mid-stride leg position. If we simply place the dummy in identical relative positions at Frames 0 and 12, we will find that the leg behaves or bends differently. This is because there are no keys before Frame 0 or after Frame 12 and the IK solutions for each are different.

One way to solve this is to create keys at Frame 1 and Frame 13 that follow our established sequence. This is the cleanest way to accomplish the desired result, but it also takes a while longer to figure out where those keys should be. A quick way to accomplish the same results is to make a snapshot of the leg from one of the middle sequences, then use this as a reference for getting the dummy into the proper position during the transition. In this case, Frame 6 is the pose that can be copied. Once we snapshot the Frame 6 mesh, we can slide it into position at Frame 0 and the leg is adjusted accordingly. Then repeat this process for Frame 12. If you play your animation at this point, you will see the front two legs behaving properly while the rest of the legs come along for the ride. (See Figure 12.)

Finishing the leg animations. The next step is to select one leg at a time and animate it according to the sequence we established earlier. I recommend that you

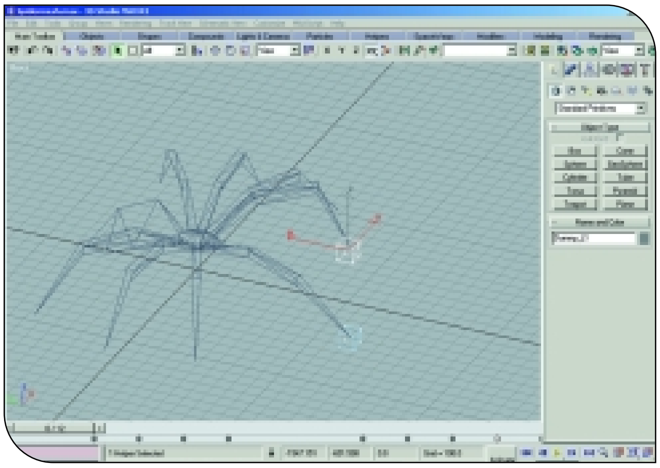
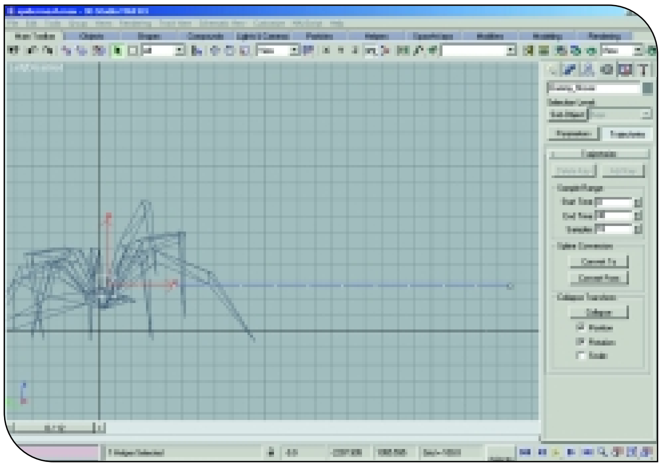



FIGURE 12 (above left). The front two legs animating in a seamless loop. FIGURE 13 (above right). The spider traveling in an unnaturally straight line.

do it in order from front to back. In our case, this would be R1, L1, R2, L2, and so on. Once you have completed one leg, always do the opposite leg right after it. After each set is done, run your animation at various speeds to check for anomalies or pops in your transition frames. You may find it necessary to adjust the stride length of some legs to compensate for the extension and compression of the leg next to it. This will avoid the legs clipping into one another, which is something that is visible even from the game's relatively distant camera.

Adding the Finishing Touches

Now that all of the legs are in motion, your spider should be looking fairly complete. However, the body will still appear to move with mechanical precision, showing no fluctuation horizontally or vertically as it travels. Adding some subtle up-and-down motion as well as some side-to-side motion will make it look much more natural. An easy way to see this represented graphically is to turn on the trajectories in Max (or a similar function if it exists in the package you're using). This creates a visible spline path that shows where each bone is passing through space. Trajectories are extremely helpful in ironing out problems and smoothing animations. Often they will show sharp jumps that are difficult to see when scrubbing the animation back and forth. By turning on a trajectory for the body, we can see that it is traveling in a perfectly straight line (see Figure 13).

Such behavior is unnatural and will draw unwanted attention to the animation. You can experiment a bit to find what suits your tastes visually. The basic idea is to add some up-and-down motion to the body as well as some rocking side-to-side. Not very much motion is needed to eliminate the robotic feel of the body. Also, as with most looping animations, you should avoid drastic or unique motions if possible: while these unique motions might look great when played once, they become very obvious and painfully repetitive during a looped sequence. The trick is to give the subject enough motion to make it look alive and natural, while avoiding any bold movements that will draw players' attention to them as the loop repeats during gameplay.

You should also go in and give the fangs a bit of motion as well. Try to avoid moving both fangs in an identical fashion. Offset the motion of one side by a frame and it will give them a more natural feel.

Another thing you can play around with is to which frame of each leg sequence you apply the Z height adjustment. This can have an effect on how the spider appears (aggressive as opposed to tentative, for example). Bear in mind that any changes may require you to readjust the transition frames.

That's it! You now have a spider walk animation that is compact and ready for use. Your dungeon can now be filled to overflowing with fast, creepy spiders that have an attitude.

FOR MORE INFORMATION

Check your backyard, garden, or behind the bookcase.

WEB SITES

The Arachnology Home Page
www.ufsia.ac.be/Arachnology/Arachnology.html

Ed Nieuwenhuy's Spiders Page
www.xs4all.nl/~ednieuw/Spiders/spidhome.htm

FILM AND VIDEO

Arachnophobia (1990) by Hollywood Pictures and Amblin Entertainment

Also, check your local video store or library video section on nature and insects.

CD-ROM

Eyewitness Encyclopedia of Nature 2.0 by DK Interactive

Where'd It Go? It Was Just Here!

Managing Assets for the Next Age of Real-Time Strategy Games

AGE OF EMPIRES II: THE AGE OF KINGS consisted of more than 40,000 game and production assets, ranging from bitmaps and textures to 3D models, sounds and music, and source code files. However, with the exception of the source code, managing game assets at Ensemble Studios has largely consisted of editing, copying, and renaming files on local and shared network drives. This process has sometimes resulted in a number of problems, including misplacement, corruption, or accidental loss of game assets. All of these problems result in effort that must be spent finding or re-creating missing assets.

With the increasing number of assets and people involved in game projects, manually maintaining game assets takes on an ever-increasing portion of the project. In order to reduce, and hopefully eliminate, this time from future game projects, Ensemble Studios decided to evaluate its own asset management needs and implement a system for storing and managing all game assets. The purpose of this article is to discuss how we translated our asset management needs into an effective asset management system for future games, and the technologies that we utilized in doing so.

Needs of the Many

During the development of previous Ensemble games, game assets were managed using a directory structure that was centrally located on a network server and copied to the user's workstation as needed. Assets were edited locally with final changes copied back up to the server or edited directly on the server itself. To indicate successive revisions of an asset, incremental numbers were sometimes added to the end of the filename.

This combination of local and server files created confusion when two users both attempted to work on the same shared server asset, or they made different local versions that were later copied back up. It was also difficult to determine which older revisions of assets were truly good enough to keep and which could be thrown away.

However, even with these potential problems, there were several big advantages to a centrally located, directory-based asset system. The first advantage was that the servers hardly ever went down. There were few times during the course of developing AGE OF EMPIRES II: THE AGE OF KINGS (AOK) when server or network problems disrupted access to game assets.

Another definite advantage was simplicity. Every user was already familiar with copying files between Windows folders. Updating an asset on the server, or adding an asset, or putting a new asset into the game was as simple as copying between Windows folders. We were able to use the pros and cons of the current directory-based asset management system to create a list of requirements for the new asset management system that built on the positives but removed the negatives.

Beyond these requirements, a new asset management system had to be able to handle a file of virtually any size, as art and sound files can range in size up to hundreds of megabytes. Another requirement was that the new system

HERB MARSELAS | *Herb is a 3D engine specialist working on the next age of real-time strategy games at Ensemble Studios, the creatively titled RTS3. The first rule of RTS3 is that you don't ask about RTS3. Drop him a line at hmarselas@ensemblestudios.com.*

Can you count the assets? Hundreds of new assets were created for the AGE OF EMPIRES II: THE CONQUERORS expansion pack, including those for this Mayan city deep in the Yucatan jungle.

Asset Management Requirements

1. Shield assets from accidental damage
2. Prevent multiple users working on an asset at the same time
3. Track all asset modifications
4. High availability
5. Simple UI (à la Microsoft SourceSafe)
6. Manage any asset regardless of size
7. Allow user to edit asset only on local workstation
8. Produce game build from current asset versions
9. Simple workflow

FIGURE 1. Ensemble's requirements for their asset management system.

should be based on serving the asset to the user's local workstation for editing. This was especially critical, as our main 3D content package, 3D Studio Max, had problems editing files across the network.

The new system also had to be capable of exporting a complete set of game-ready assets from those under asset management. This would remove the onus from the project teams of trying to verify that they really had copied the very latest version of an asset into the appropriate game directory. As long as the latest revision of the asset was in the asset manager, they could be assured it would get in the game.

The final requirement was for a simple workflow system to help the art team keep better track of where assets were in the art pipeline. The workflow system would have three nodes, allowing an asset to be tracked from prototype, to ready for game use, to finalized.

Having established the requirements for the new system (Figure 1), we then faced the looming question: Build or buy?

A Single Solution?

Because the programming team was already using Microsoft SourceSafe 6, the first task was to examine the viability of using it as an asset management system for the whole team. While SourceSafe offers a good user interface, a stand-alone version, and an API to create tools to interface with it, a number of concerns arose immediately. The biggest of these were issues of dealing with files, and even a moderate number of users.

TABLE 1. NT-based asset management products.

COMPANY	PRODUCT	REPOSITORY	API?	ASSET SERVER?	LIMIT ON TOTAL BYTES OF ASSETS?
Bulldog	Two.Six	Relational Database (RDBMS)	Yes	Yes	Limited only by RDBMS
eMotion	Cinebase3	RDBMS	Yes	Yes	Limited only by RDBMS
Filemaker	Filemaker Pro	RDBMS	Yes	Yes	Limited only by RDBMS
Microsoft	Visual SourceSafe 6	File-based	Yes	No	Yes: 4GB
Merant	PVCS	Proprietary database	Yes	No	Not documented
NxN	Alienbrain	RDBMS	Yes	Yes	Limited only by RDBMS

TABLE 2. Is there a single solution?

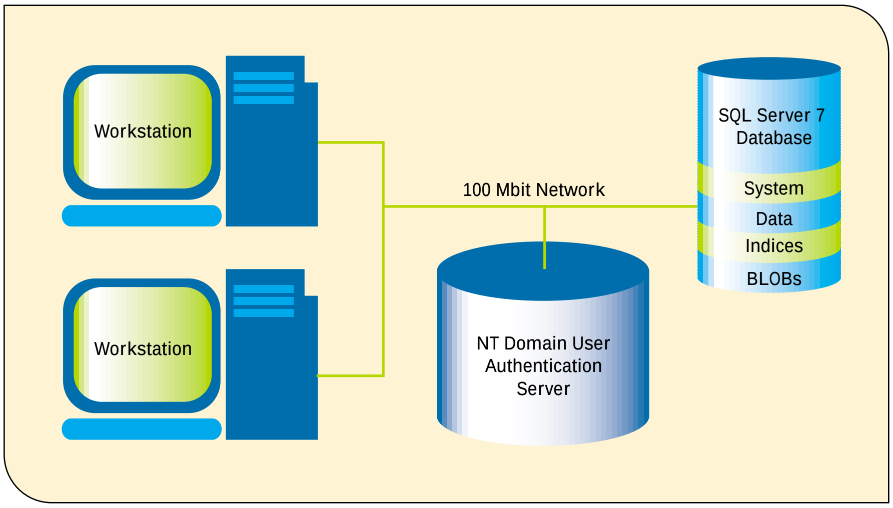
PRODUCT	PROS	CONS	BOTTOM LINE
SourceSafe	• Simple, easy-to-use interface • Integrated with Microsoft Visual Studio and C++ • Programmers already use it	• File-based repository • 4GB data limit • Can't handle really big files • Gets confused when a single user checks out files on multiple workstations	• Good for teams working on assets < ~10MB in size who don't want or need a back-end server • Competent user interface is easily picked up by programmers and nonprogrammers alike
PVCS	• Integrated with Microsoft Visual Studio and C++	• User interface not friendly or intuitive • Proprietary database • Has trouble handling some types of Visual C++ projects	• Good for teams working on assets < ~10MB in size who don't want or need a back-end server, but who need more than 4 GB of asset space • Programmers may not be happy about the Visual C++ project problems, and everyone may hate the user interface
Alienbrain	• Large API • Uses relational database (RDBMS) for asset repository	• Large API • Stand-alone UI, but no integration with Visual C++ or 3D Studio Max	• Good for teams that need a back-end asset repository but don't have back-end programming, design, and architecture experience • The large API may be overwhelming

With only 15 programmers using it, our SourceSafe system was having performance and consistency problems that resulted in a number of hours spent each month in maintenance and recovery. Working with Microsoft support, we found that the problems we were experiencing affected some number of SourceSafe sites with no discernable cause. Other problems with SourceSafe included a severe performance problem when attempting to check in large assets (anything bigger than about 10MB), even when the files were stored directly, and there was confusion when a user had files checked out on more than one workstation. Although we chose not to use SourceSafe, it did have two features that we could not ignore: its simple user interface and seamless integration with Microsoft Visual C++. These became our guidelines for usability in selecting an asset management system (see Tables 1 and 2).

In our quest to try to keep the whole team using a single asset management tool, we also evaluated Merant PVCS. With PVCS we came to some of the same conclusions that we had reached with SourceSafe. While PVCS is a good tool for simple programming projects, we found it to have many of the same shortcomings as SourceSafe for game asset management. It also lacked a good, intuitive stand-alone UI.

Finally, we looked at NxN's Alienbrain. Alienbrain is more like an asset management toolbox than an off-the-shelf asset manager. This meant that if we did use it, we couldn't just drop it in and go. We would have to learn how to use its interfaces, then build an asset management system on top of them.

Three other asset management products we looked at briefly were eMotion's Cinebase 3, Bulldog Two.Six, and Filemaker Pro. In general, these are all compe-



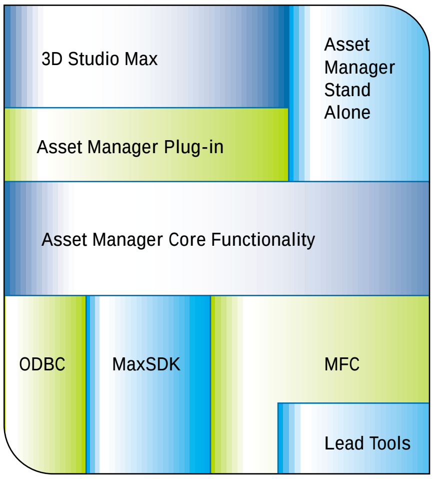


FIGURE 2 (above). Back-end network and server configuration. FIGURE 3 (left). Integration of client-side components.

tent asset management tools. However, they suffer from the same issues that plague the other products we reviewed in greater depth — a lack of front-end integration and workflow. They are potentially more powerful back-end solutions depending on your need, but in the end a lot of time will be spent creating a custom solution relying on their individual APIs.

The Final Solution

The biggest issues for all these tools, however, are the lack of front-end integration and even the simplest type of workflow. If we were to use any of these asset managers, we would have to learn their APIs and then spend the time creating a workflow system and integration with front-end tools such as 3D Studio Max.

Because we already had expertise in the design and architecture of large-scale databases, we decided to spec out how long the implementation would require if we created an asset management system from scratch. We estimated the time needed to create our system would be approximately six man-weeks of programming time. This did not include time for testing or for major new requirements or features that cropped up during development, testing, and deployment.

The asset management system we ultimately implemented consisted of four major components. The largest component was the relational database on the back-end to track and manage the assets, and the client-side data access layer to it. On the client side, there were two user interface components. One was a plug-in (Figure 3) to integrate the asset management functionality seamlessly with Max, and the other was a stand-alone front end for those users. The final component was an exporter used to create game builds from the latest versions of assets.

Back-End and Client

On the back-end, we chose to use Microsoft SQL Server 7 as the central database (Figure 2). Oracle or Sybase would have worked just as well, and may be required if we increase the number of users significantly. However, SQL Server 7 running under Microsoft Windows NT 4.0 is enough computing power to handle our current and future needs at this time.

We also decided to store the asset revisions directly inside SQL Server as binary large objects (BLOBs). We could have stored them on a network drive, but we felt that SQL Server could stand the additional load, and storing them in the database provided additional security.

To improve performance of the database, especially with the large number of assets stored inside it, the database server was configured with two ultrawide SCSI controllers. Each controller then supports two ultrawide SCSI hard drives. This configuration allowed us to place the database system files and logs, the small asset management data, the large BLOBs, and the index data on separate drives. This improves performance by allowing the database to spread its access patterns across all four drives. Database security itself is handled directly through the Windows NT domain user authentication system. This means that granting users database access is as simple as adding them to one of the existing NT domain groups.

On the client side, we built a data access layer using ODBC. This gave us several advantages. First, it's simple to maintain. It's also easy to learn. Moreover, there is no dependency on bound data controls (such as MFC), which allows us to actually build an in-game connection to the back-end SQL Server by just adding the additional link to the ODBC libraries.

All of the asset manager's client functionality was then created in a single layer on top of ODBC and other core technologies (Figure 3). The specifics of the Max plug-in and the stand-alone user interface are then abstracted into separate files on top of the core functionality. This creates a system where all of the code is completely shared through most of the two separate user interfaces.

Regardless of whether the user accesses the asset manager through Max or the stand-alone user interface, both systems present the asset manager to the user with just a Windows Explorer interface (Figure 4). Because all of the users are already familiar with navigating a tree structure, this has significantly reduced the users' learning curve when using the system.

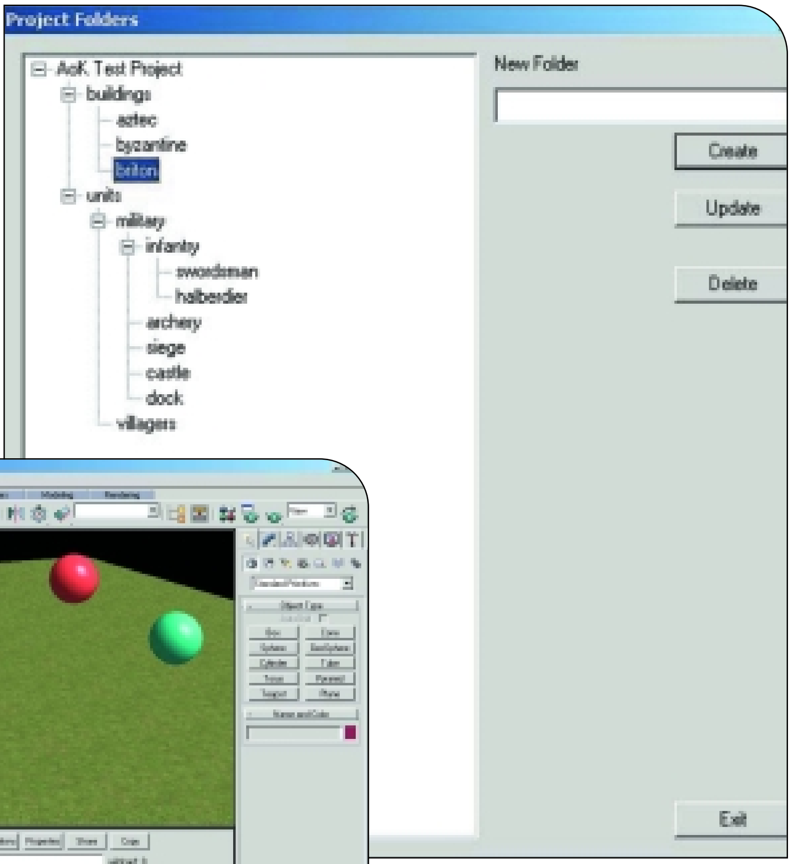
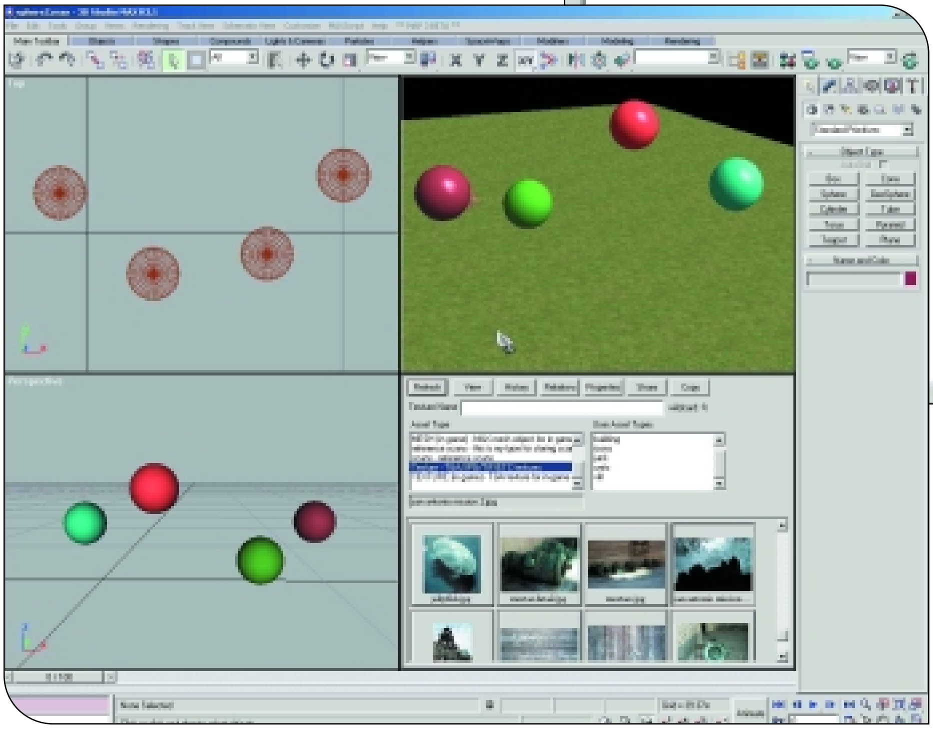


FIGURE 4 (right). Creating a project folder using Windows Explorer interface. FIGURE 5 (below). 3D Studio Max with RTS3 in-game extended viewport (upper right) and bitmap browser extended viewport (lower right).



Keep the Artist in Max

During the development of AoK, the art process consisted of a number of individual steps. Once an artist had created a unit for in-game use, it then had to be rendered out, postprocessed, then added to the game by a lead artist or designer. This meant that it could be a long time between the time artists worked on an asset and when they actually saw it in the game.

As we started implementing the asset management system, our mantra became "keep the artist in Max." In other words, give the artists everything they need to create, manage, and view their models and textures in the game directly in Max. The only time they should have to leave Max is to run Adobe Photoshop, which they can also launch from a button in Max.

This mantra led to the creation of two additional Max plug-ins to support our asset management system. The first was a texture/bitmap browser that allowed the artist to search, view, check out, and use any bitmap asset stored in the asset manager (Figure 5). The texture browser was built as a Max extended viewport plug-in using MFC and Lead Tools imaging control. The Lead Tools provided us a very powerful yet simple-to-use ActiveX component that could be used directly in the MFC dialog box that formed the basis of the texture browser.

The second plug-in was also a Max extended viewport — it was actually the entire game running inside Max as a viewport. This plug-in was relatively simple to create by making a version of the game that built as a .DLL rather than an .EXE file. Some additional code was required in the window handling code to compensate for the fact that the game was now running as a child window and had to interact with the Max input system.

A small amount of functionality was also required to allow the artist to specify which of the objects in a Max scene should be put into the game. Instead of creating a cumbersome communication system between Max and the game, the artist's scene is exported to a file in a format that the game can load. The game is then told to load the file and display its

contents. Because this is the whole game engine running inside Max, the artist can examine the model in the context of other existing game assets and scenarios. As complicated as getting the game engine itself to run within Max was, creating both extended viewports was very straightforward compared to managing Max files and seamlessly integrating with Max itself.

Max files themselves might initially be located in any local or server directory, and they may refer to texture files that exist in any of the Max texture paths. Adding a Max file to the asset manager meant moving the file to its managed directory, then scanning the file for any texture files and moving those to the same directory as well, and finally updating the paths of textures referenced by the Max file.

If the Max file contained in-game models, not only was the Max file added to the asset manager, but an in-game version of the file was generated and stored in parallel. This allowed the back-end export process used to create the game build to be much simpler. It just had to copy data out of the database and store it in files.

Every texture also generated one or two additional files on check-in. A thumbnail image was created and stored to support the texture browser, and textures used in-game automatically generated a texture in the in-game format.

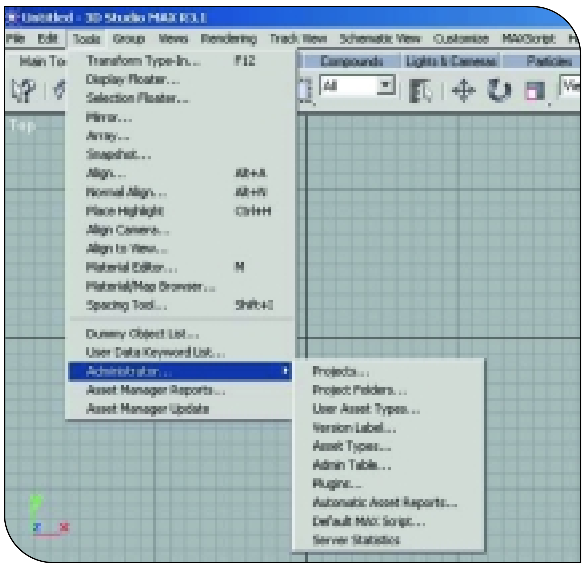


FIGURE 6. 3D Studio Max menu integration.

Integrating with Max

Integrating directly into the Max user interface and menu system was one of the hardest challenges we faced in creating a seamless asset management system (Figure 6). Unlike the Microsoft Visual C++ IDE, Max does not have a well-defined interface for asset management tools to plug into. Also, it's not possible to create menu items in Max using the Max SDK or MaxScript.

Because we wanted to make the integration as seamless as possible, we had to rely on Win32 programming to manipulate the main Max window and menu system directly. Although this appeared to be a complicated solution, it gave us the flexibility we desired to create new menu entries, and the ability to override and enhance existing Max functionality.

To facilitate this integration, we used a Max general utility plug-in (GUP) to glue Max and the asset management system together. The GUP is one of a number of DLL-based plug-ins that Max supports for modifying or extending existing functionality. Before reading the following explanation of how we integrated the GUP plug-in directly with the Max menu system, you may want to download the source code from the *Game Developer* web site at www.gdmag.com.

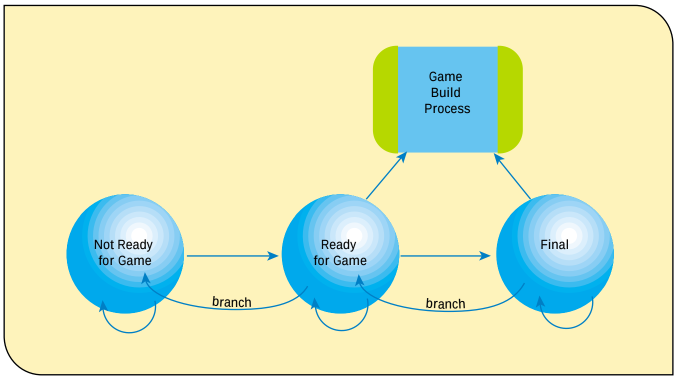
The Max plug-in architecture is based on deriving developer-implemented classes from base Max C++ classes. In this case, our `MaxUIModGUP` class is derived from the Max GUP class. Max identifies plug-ins in two ways: through their file extension (.GUP for a GUP plug-in), and with a simple class factory called `ClassDesc`.

When this DLL plug-in is built, we change the file extension from .DLL to .GUP and place it in the MAX PLUGINS\ directory. As Max scans for each set of plug-ins it recognizes, it knows that this plug-in is at least used as a general utility plug-in. Once loaded, Max uses four simple functions to interrogate the DLL. `LibDescription` returns a simple text description of the plug-in. `LibNumberClasses` returns the number of class factories (or `ClassDescs`) in the plug-in. `LibVersion` is the version of Max that the plug-in will work with. And most importantly, `LibClassDesc` returns an instantiation of our own derived version of `ClassDesc` called `MaxUIModClassDesc`. Max can instantiate our `MaxUIModGUP` class using the `MaxUIModClassDesc::Create` member.

In the case of some plug-ins (for example a viewport plug-in), this class factory could be called multiple times. For a GUP it is only called once at startup. This is why we can simplify the code somewhat by using global variables to store our flags and state information.

Once instantiated, the `MaxUIModGUP::Start` member function is called. `MaxUIModGUP` has access to the main Max window handle using the inherited member function `MaxWnd`. Once we get the window handle, we can subclass it with our own message handler (`SubclassWndProc`) and return success. Subclassing the window ensures that we get the Max window messages before its message handler does.

In implementing our scheme to add new menu entries and enhance existing functions, we only care about two Windows messages: `WM_INITMENU` and `WM_COMMAND`. `WM_INITMENU` is sent to the window when a menu is about to become active. This allows us to look for the main menu, and modify its functionality. However, we must be careful to modify only the main menu, and then to modify it only once.



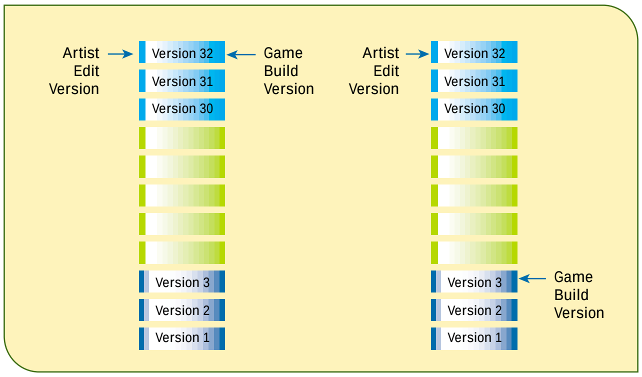


FIGURE 7 (above left). A simple three-node workflow system with branching. FIGURE 8 (above right). Example of unbranched (left) and branched (right) assets.

Because Max creates a number of menus, we use `GetMenuItemCount` and `GetMenuString` to make sure the menu we're getting is the main menu. Once we've ascertained that we have the correct menu, the `modifyMenu` function inserts two new entries into the file menu. For this sample, we're adding a menu option that will force a complete redraw, plus a separator to make the menu look pretty. In the end, `DrawMenuBar` is called to make sure that the menu is properly updated when it draws.

Back in the `SubclassWndProc` function, we need only add the `WM_COMMAND` case to look for the new menu entry we created (`IDC_MAXMENUMOD`), and process it accordingly. One additional piece of functionality that has been added to this `WM_COMMAND` handler is a wrapper for the Max File Open menu entry. This code stores off the current Max project filename, calls the default handler to open a new file, and then displays a message box to inform you if a new file has been opened.

Overriding the File Open function in this way may seem a bit dangerous, because the File Open menu entry might not be 0x9c43 in a future version of Max. Unfortunately, there's no other viable way to add a very important piece of asset management functionality: detecting when a user opens a file that he or she doesn't have checked out from the asset manager. While you can register a callback using the Max function `RegisterTimeChangeCallback` to determine if the current filename has changed, you don't receive the notification until after the file is opened. The Max notification system may look like another alternative, but it's just that: a notification. You can't stop or change something already in progress.

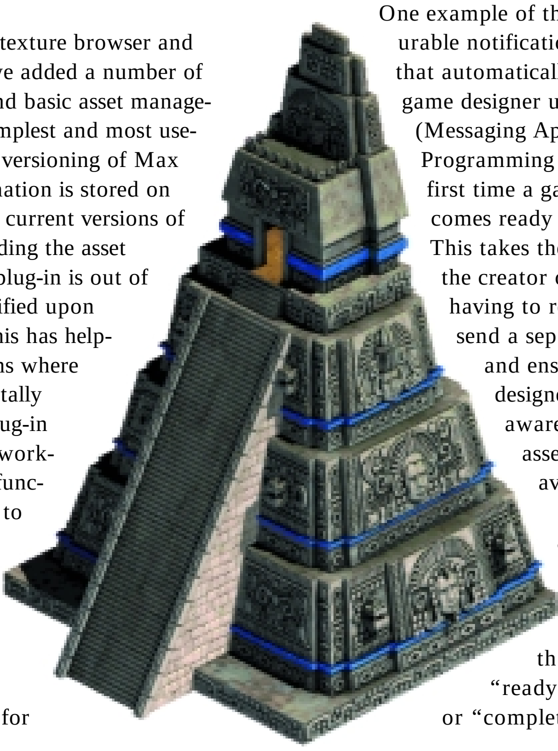
The asset manager needs to have a priori access to File Open requests for several reasons. If the user doesn't have the file checked out, it will be read-only. It might also not be the latest version of the file. Either way, if the file is under asset management the latest version of the file needs to be copied down to the user's workstation and made writable if someone else doesn't already have it checked out.

Beyond Asset Management

In addition to the texture browser and game viewport, we added a number of other features beyond basic asset management. One of the simplest and most useful features was the versioning of Max plug-ins. File information is stored on the server about the current versions of Max plug-ins, including the asset manager itself. If a plug-in is out of date, the user is notified upon starting up Max. This has helped in many situations where the user has accidentally installed an older plug-in when reinstalling a workstation. Additional functionality was added to allow the asset manager to update itself automatically when new versions are posted internally. This has eliminated the need for the user to worry about keeping track of the latest version and its location. All users need to do to get the latest version is to restart Max.

The three-node workflow system helped us understand the state of game assets more clearly (Figure 7). New assets are marked "not ready for game." When an asset is ready to be exported to the game, it becomes "ready for game," and then finally "final." An asset can stay in any node as long as is necessary.

One of the benefits of using SQL Server directly was that we could create stored procedures to enhance asset workflow. One example of this is a configurable notification system that automatically e-mails the game designer using MAPI (Messaging Application Programming Interface) the first time a game asset becomes ready for the game. This takes the burden off the creator of the asset of having to remember to send a separate e-mail, and ensures that the designer is kept aware of when assets become available.



There is also a branching system so that an asset that is either "ready for game" or "completed" can be branched to the next lower node. This allows the artist or designer to leave the last version that they know is good enough for the game to continue to be exported, while allowing them to go back and potentially make major revisions. Once the user is ready for the latest version of the asset to be exported again, the asset is unbranched (Figure 8).

Asset types were added to allow the teams to classify new assets added to the system. Some of these asset types, such as "Mesh (in game)" or "Texture (in game)" are backed by code in the asset manager that performs special functions. In these two cases, in-game versions of the assets are created and checked in whenever the original assets themselves are checked in.

Future Work

While our asset management system has solved a number of problems that we have experienced in managing game assets in the past, several areas still need additional work. The hardest area that we have yet to address is the issue of the "build machine" — the versioning of the tools themselves. We've already run into this problem once in creating a patch for one of our previous games almost a year after it shipped. The original game was compiled using Microsoft Visual C++ 5. Our current compiler is Visual C++ 6 with Service Pack 3.

Getting the game to finally compile correctly with the latest version of the compiler took several days. While the new compiler certainly improved the code quality and allowed us to find some problems that the old compiler had missed, we don't know what additional problems the new compiler might have introduced.

It seems that the only way to fully version-off the tools that were used to create the game is to set a workstation in the corner that contains the tools and the shipping version of the source code. Any other solution that involves storing the program install disks away and having to reinstall old software probably will not be utilized.

One of the other problems to solve is being able to regenerate all the in-game versions of assets from the original versions stored in the asset manager. Although we try to make sure that we never break existing game assets, there usually comes a time (sometimes more than once, unfortunately) where all of the art, sound, or levels for the game need to be reprocessed. With the original data, and the information on how it was exported for the game stored in the asset manager, it's possible to write a batch system that can perform this reconversion. One of our next goals is to actually implement this using the Max DCOM sample code to create our own batch system to reprocess assets as necessary.

Summary

In the end, it would have been easier if there were an off-the-shelf asset management solution that met our needs right out of the box. Those asset management systems directed towards game and content creation all seem competent in delivering basic asset management functionality, but they are sorely lacking in the areas of front-end and third-party integration and workflow.

For us, developing an asset management system completely in-house was well worth the effort. We ended up with a system that met our needs, and because we fully understand the design and architecture, we can continue to add functionality as needs arise.

The total staffing time was about eight full-time man-weeks of programming, and three part-time weeks of testing. The programming time did run over by a couple of weeks, but this was mainly due to adding new features during development. It should also be noted that the programming time would have taken significantly longer if we hadn't already had back-end server design and implementation experience in-house.

Overall, the biggest problems revolved around our inability to modify and integrate with the Max user interface system more simply. The whole Max user interface needs to be based on MaxScript, and MaxScript itself needs to be much more powerful in order to do what we need to do. An even better solution would be if Max provided a specified asset management interface similar to Microsoft's Visual C++, thereby allowing third parties to create asset management solutions for it.

ACKNOWLEDGEMENTS

Creating an effective and useful asset management system was a team effort. Thanks to everyone involved throughout the process for remembering those little details. Thanks also to everyone at Ensemble Studios for reviewing this article.

FOR MORE INFORMATION

MSDN

Felder, Ken. "Microsoft Visual SourceSafe OLE Automation." Microsoft Developer Network. Microsoft Corp., October 1995.
http://msdn.microsoft.com/library/techart/msdn_vssole.htm

WEB SITES

Bulldog Two.Six
www.bulldog.com

Discreet 3D Studio Max
www.discreet.com

eMotion Cinebase3
www.emotion.com

Filemaker Pro
www.filemaker.com

Lead Tools
www.leadtools.com

Microsoft SQL Server
www.microsoft.com/sqlserver

Microsoft Visual SourceSafe 6
http://msdn.microsoft.com

Oracle
www.oracle.com

Sybase
www.sybase.com

Merant PVCS
www.pvcs.com

NxN Alienbrain
www.alienbrain.com

Return

Ion Storm's DEUS EX

GAME DATA

PUBLISHER: Eidos Interactive

NUMBER OF FULL-TIME DEVELOPERS: Approx. 20: 1 of me, 3 programmers, 6 designers, 7 artists, 1 writer, 1 associate producer, 1 tech.

NUMBER OF CONTRACTORS: Approx. 6: 2 writers, 4 testers.

LENGTH OF DEVELOPMENT: 6 months of preproduction and 28 months of production.

RELEASE DATE: June 23, 2000

INTENDED PLATFORMS: Windows 95/98/NT/2000 plus third-party Macintosh and Linux ports

CRITICAL DEVELOPMENT HARDWARE: Ranged from dual Pentium Pro 200s with 8GB hard drives, to Athlon 800s with 9GB fast SCSI, and everything in between. More than 100 video cards were cycled through during development.

CRITICAL DEVELOPMENT SOFTWARE: Visual Studio, Lightwave, Lotus Notes.

NOTABLE TECHNOLOGIES: Unreal engine and associated tools such as UnrealEd and ConEdit (our proprietary conversation editor).

Fictionally, DEUS EX is set in a near-future version of the real world (as it exists if conspiracy buffs are right). For some real shorthand, call it "James Bond meets *The X-Files*."

Conceptually, DEUS EX is a genre-busting game (which really endeared us to the marketing guys) — part immersive simulation, part role-playing game, part first-person shooter, part adventure game. It's an immersive simulation game in that you are made to feel you're actually in the game world with as little as possible getting in the way of the experience of "being there." Ideally, nothing reminds you that you're just playing a game — not interface, not your character's back-story or capabilities, not game systems, nothing. It's all about how *you* interact with a relatively complex environment in ways that you find interesting (rather than in ways the developers think are interesting), and in ways that move you closer to accomplishing your goals (not the developers' goals).

It's also a role-playing game in that you play a role and make character development choices that ensure that you end up with a unique alter ego. You make your way through a variety of minute-to-minute gameplay experiences (which add up to a story) in a manner that grows naturally out of the unique aspects of your character. Every game system is designed to differentiate one player-character from another, and to allow players to make decisions that reflect their own biases and express character differences in obvious ways in the game world.

It's a first-person shooter because the action unfolds in real time, seen through the virtual eyes of your alter ego in the game world. To some extent, your reflexes and skill determine your success in combat. However, unlike the typical FPS, DEUS EX doesn't force you to shoot every virtual thing that moves. Also unlike the average FPS, in which gameplay is limited to pulling a virtual trigger, finding blue keys to open blue doors, and jumping to reach seemingly inaccessible locations, DEUS EX offers players a wide range of gameplay options.

And finally, DEUS EX is like adventure games in that it's story-driven, linear in narrative structure, and involves character interaction and item accumulation to advance the plot. However, unlike most adventure games (in which you spend the bulk of your time solving clever puzzles in a search for the next static, but very pretty, screen), DEUS EX asks players to determine how they will solve game problems and forces them to deal with the consequences of their choices.

DEUS EX combines elements of all of these genres. But more important than any genre classification, the game was conceived with the idea that we'd accept players as our collaborators, that we'd put power back in their hands, ask them to make choices, and let them deal with the consequences. It was designed from the start as a game about player expression, not about how clever we are as designers, programmers, artists, or storytellers. Which leads naturally to a discussion of having clear goals — the first thing I think we did right.

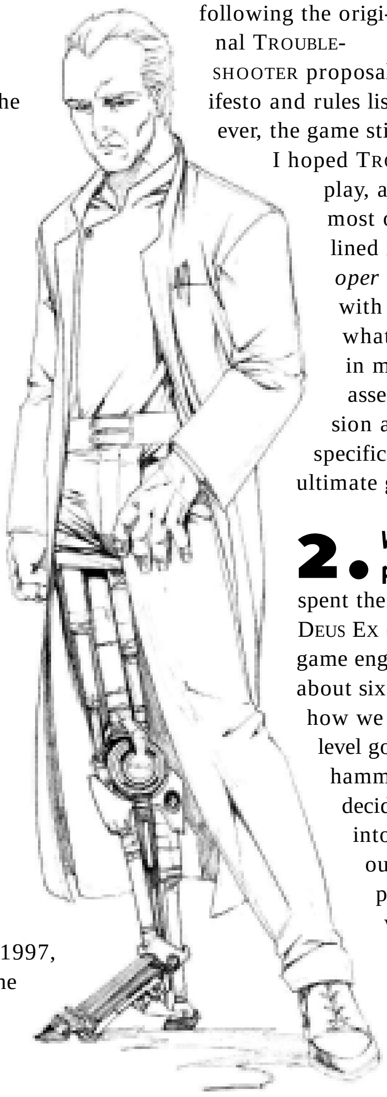
WARREN SPECTOR | *Warren runs Ion Storm's Austin, Texas, office. He produced and directed the action/RPG DEUS EX. Prior to that, he produced ULTIMA WORLDS OF ADVENTURE: MARTIAN DREAMS, ULTIMA VII PART 2: SERPENT ISLE, UNDERWORLD 1, UNDERWORLD 2, WINGS OF GLORY, SYSTEM SHOCK, and others for Origin and Looking Glass Technologies. You can reach him at wspector@ionstorm.com.*

The Deus Ex player's alter ego, J.C. Denton, strikes a heroic pose.

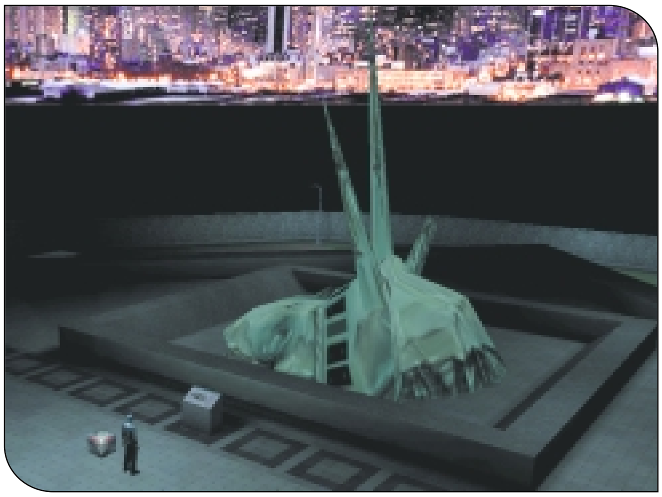
What Went Right

1. A clear high-level vision. It's pretty self-evident that you can't achieve goals if you're not clear about what they are. We knew with a high degree of confidence what kind of game we wanted to make. This was possible for two reasons. First, DEUS EX is a natural outgrowth of work done by and in some cases with the late, lamented Looking Glass Technologies. We were inspired as well by games made at Valve, Origin, and a host of other places. Many of the things we wanted to do were a reaction to things they (or we) didn't do, didn't do well, or couldn't do at all in earlier games. We weren't building from scratch, but rather building on a foundation already laid for us.

Second, and on a personal level, DEUS EX is a game I've been thinking about since right around the time UNDERWORLD 2 shipped. I've tried to get a game like this started several times (as TROUBLESHOOTER at Origin; in some respects, as JUNCTION POINT, for Looking Glass). Those games didn't happen for a variety of reasons, but I never stopped thinking about them and, despite the failure of those games to reach production, they laid much of the conceptual groundwork for DEUS EX. The lesson here is that if there's a game you really want to make, don't give up on it. Someone will be foolish enough to give you the money eventually.



Several years passed. I left Origin to go work for Looking Glass, but TROUBLESHOOTER stayed on my mind. In the fall of 1997, before Ion Storm entered the DEUS EX picture, I drafted a manifesto — a description of an ideal game — and also a set of "rules of role-playing." Much of that material ended up in an article published in *Game Developer* ("Remodeling RPGs for the New Millennium," February 1999), which is still available online on Gamasutra.com.



ABOVE. Real-world spaces, such as the Statue of Liberty in New York City, can be compelling game spaces, but offer unique challenges to game developers.

The details of DEUS EX — plot, character lists, game system designs and so on — changed radically in the years following the original TROUBLESHOOTER proposal and writing my manifesto and rules list. Conceptually, however, the game still plays much the way I hoped TROUBLESHOOTER would play, and it definitely fulfills most of the ideals I had outlined in that *Game Developer* article. Quite simply, with a solid concept of what we wanted to achieve in mind, we were able to assess every design decision and every game system specification in light of our ultimate goals.

2. We didn't skimp on preproduction. We spent the first six months of DEUS EX (before we licensed a game engine), with a team of about six, just thinking about how we could turn our high-level goals into a game. We hammered on the setting and decided to move the game into the near future to buy ourselves some room to play around — the real world, as we quickly discovered, was very limiting. Ultimately, we settled on a conspiracy-oriented background.

We did a vast amount of research into "real" conspiracies — the Kennedy assassination, Area 51, the CIA pushing crack in East L.A., Dwight Eisenhower's UFO connection, and of course Freemasons tunneling below the Denver airport and building abducted-baby cafeterias for alien invaders at George Bush's direction. Only a fraction of this stuff ended up in the game, but it gave us a peek into the minds of conspiracy buffs that was both scary and useful.

We worked on back-story stuff so we'd know what was going on in the world, even in places the player never got to visit. Some of this stuff may come to the forefront in DEUS EX *2* but, for DEUS EX, it was just a way of making sure we knew enough to include the kinds of small details that make a fictional world convincing.

We also created a cast of more than 200 characters, many of whom didn't yet have specific roles in the game. Ultimately, this list proved to be both a help and a hindrance to designers as they fleshed out the missions. Characters sometimes suggested missions or subquests, but just as often ended up being filler we were reluctant to cut, even though their missions or story purposes changed during our storyline-focusing passes.

We hammered on game systems. We conceived a skill system that didn't depend on die-rolls or tracking skills at a fine level of granularity. We came up with a system

LEFT. A Hong Kong temple, modeled from actual photographs. RIGHT. Either a failed stealth attempt or a frontal assault — the choice is up to the players in DEUS EX.

of "special powers" (nanotech augmentations) that differentiated the player character from ordinary humans. We designed a conversation system with some cinematic elements and some elements borrowed from console RPGs. We mocked up 2D inventory, skill, and augmentation upgrade screens, map screens, even a text editor so players could take notes. We conceived several player reward systems, including skill point awards, augmentation upgrades, weapon availability timelines and tool/object availability timelines.

By March 1998, we had 300 pages of documentation and thought we knew everything we'd needed to know to make a game. Were we ever wrong. In the time between March 1998 and our Alpha 1 deadline of April 1999, that 300-page document mushroomed into more than 500 pages, much of it radically different from what we thought of and wrote initially. Clear goals and a detailed script are all well and good, but goals change, thinking changes, and game designs have to change, too. Which leads nicely into the next thing that went right.

3. Recognizing that game design is an organic process. Why did our thinking and goals change? There were lots of reasons. First, new people joined the team, with new ideas. Our staff grew from six people to roughly 20. I hired a bunch of people, of course, but we had the added excitement of integrating an entire art team assigned to us, in Austin, by an art director a couple of hundred miles away in Ion Storm's Dallas office. As we brought on new people, we found ourselves to be a team of hardcore ULTIMA geeks, hardcore shooter fans, hardcore immersive sim fans, strategy game nuts, and console gamers. Some of our new team members proved to be "maximalists" — wanting to do everything, special-case lots of stuff, and stick as close to reality as possible. Other team members proved to be minimalists — wanting to include fewer game elements but implementing them exceptionally well, in ways that could be universally applied rather than special-cased.

Also, we made a point of letting select friends and colleagues play the game at various points along the way. We were interested in well-reasoned opinions from folks who understood the kind of game we were making intimately and who had a handle on the development process that was at least as good as our own. With all the new folks contributing and all the feedback from our chosen critics, well, let's just say we had some interesting debates at Ion Storm, Austin. Out of those debates new ideas arose, and the game changed as a result.

Technology forced design changes, too. It took time to become familiar with the Unreal engine. I wish I could say we uncovered all its potentials and limitations quickly, but we didn't. Months of experimentation were necessary to reveal how best to do things in Unreal and what things not to do at all. When we stopped playing with Unreal and actually started working with it (roughly six to nine months after we got our hands on it), lots of ideas we'd come up with in the abstract didn't work quite as well in reality.

A third area that influenced the changing nature of the game's design was when the game systems didn't work as we intended them to. We quickly found that descriptions of game systems are no substitute for prototypes and actual implementation. We prototyped every game system as it was documented relatively early on. We also built some test missions, not quite early-on enough, but still early.

These test systems and missions revealed gaping holes in our thinking, or things that we thought would be true and which turned out not to be true at all. For instance, once implemented, our augmentation and skill systems proved dry and rather dull, despite looking really good on paper. I thought the tension of standing outside a locked door, not knowing if a guard was going to show up while you picked the lock, would provide sufficient excitement. I thought knowing you could leap across a chasm because you had the Jump augmentation at Tech Level 3, and opening up new paths through maps that were inaccessible to players without that augmentation, would be cool enough to keep players interested.

As it turned out, when Gabe Newell from Valve came down and played our prototype missions, he correctly identified the utter lack of tension in our skill and augmentation use as they were written up in the design document and ably implemented by the coders. The worst was confirmed when Marc LeBlanc, Doug Church, Rob Fermier, and other friends from Looking Glass Studios and Irrational Games played the proto-missions and came to the same conclusions — actually using skills and augmentations revealed things that simply thinking about them could never have revealed. We took this criticism, and with it in mind, lead designer Harvey Smith revised the skill and augmentation systems pretty thoroughly, increasing the tension level, providing new rewards, and allowing players to think and make informed decisions. None of this would have happened without the prototype missions and some harsh (but fair) criticism they elicited.

Another big reason for changes from our original design document was our realization that the idea of a real-world RPG, with real-world locations and real-world weapons, was cooler in some ways than it turned out to be on the screen. Not to put too fine a point on it, but DEUS EX became a lot less realistic as time went on. When we started building places like the Statue of Liberty, a few square blocks of New York City, the White House, Parisian streets, and so on, we found that most of the real world is not all that interesting as a gaming environment. Basically, hotels and office buildings aren't great game spaces. We also found it difficult to live up to people's expectations of places they've actually been. We began to hear comments like, "That's not what the inside of the Statue of Liberty looks like. I've been there. I know." We created an object-rich environment, only to hear things like, "Hey, why can't I use that telephone to call anyone I want whenever I want?" and had to cut some objects whose real-world functionality we couldn't capture in the game.

Finally, we had to ask ourselves whether human non-player characters (NPCs) are interesting enough to carry an entire game. We were about a year into development when designers and artists balked at a game entirely about human beings. Movies don't need nonhumans to be cool but the same cannot be said, apparently, for games. People want monsters and bad guys. The feeling was so pervasive that it changed my thinking completely. The original design spec called for a couple of robots, but the team demanded that they be made a more important part of the landscape, and we introduced genetically manipulated animals and some alien-looking creatures. (Luckily, our game fiction supported all of this.) The game benefited, but this was a radical change from the original plan.

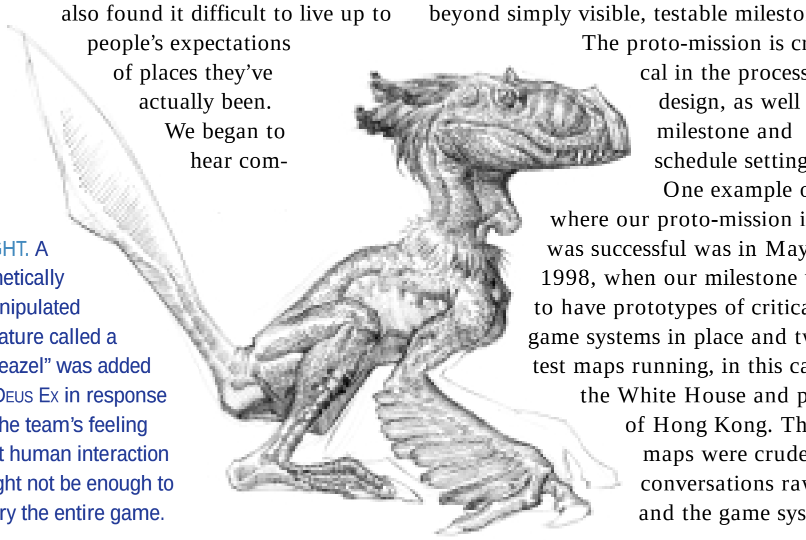
4. Creating "proto-missions."

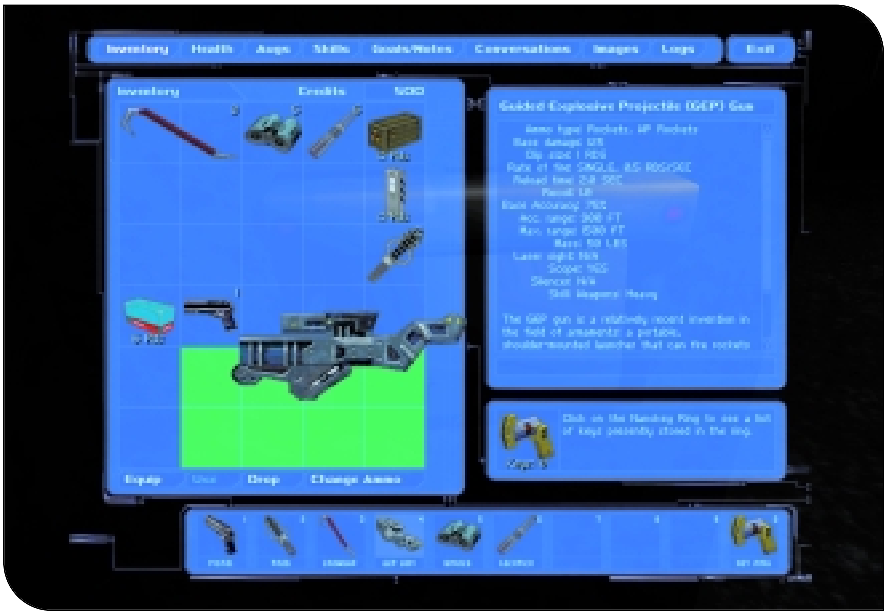
It's a truism that milestones should be testable, showing visible progress, whenever possible, and we lived up to that standard. We could always pull a version together, always show off for press or our publisher. Most importantly, we always knew where we were (even if that knowledge was sometimes painful). But the proto-mission idea is something beyond simply visible, testable milestones. The proto-mission is critical in the process of design, as well as in milestone and schedule setting.

One example of where our proto-mission idea was successful was in May 1998, when our milestone was to have prototypes of critical game systems in place and two test maps running, in this case the White House and part of Hong Kong. The maps were crude, the conversations raw, and the game systems hacked, but we could see — and show — the potential. To our advantage, we resisted the temptation to do just the stuff we knew would work and the stuff that would look the prettiest, and prototyped new, risky stuff first. Conversation, interface, inventory, skills, and augmentations were all at least hacked in so we could see them in action. The White House was likely to prove our toughest map challenge, so we built it first. (Almost unbelievably, I missed what may have been the riskiest, most critical game system in all of our early prototyping, NPC AI. I should have insisted on early prototyping of our AI but I didn't.) With the proto-mission system, we could immediately see some of the limitations of our technology. For example, we had some serious speed problems with areas as big as the White House and Hong Kong. After this, we knew we'd have to break maps up into small pieces. And we began to suspect, though I couldn't quite embrace the idea, that we'd eventually have to cut maps and missions from the game — most notably the White House.

In May 1999, we had a milestone calling for the delivery of the first two missions of the game, playable start to finish. All of our game systems were implemented (not hacked) as originally documented. You could start a game, create a character, upgrade skills, solve problems in a variety of ways, manipulate inventory, acquire augmentations, talk to NPCs, get and accomplish goals, save your game, and so on. To the team's chagrin, I had a tendency to call this the "Wow, these missions suck" milestone. It was around this time when Gabe Newell came for a visit and gave us our wake-up call, and where we all went, "Ulp! We have a lot of work to do." Our earlier demos had shown the potential of what we were doing. This demo showed us how far we had to go before we reached that potential.

This milestone also benefited us in that it showed us all the steps necessary to create a mission, and revealed the elements that really made the game work. That knowledge allowed us to go through our 500-page design document and cut everything that was extraneous, winnowing it down to a svelte 270 pages. Less game? Not at all. What was left was the best 270 pages — the stuff that worked. "Less is more"


RIGHT. A genetically manipulated creature called a "Greazel" was added to DEUS EX in response to the team's feeling that human interaction might not be enough to carry the entire game.

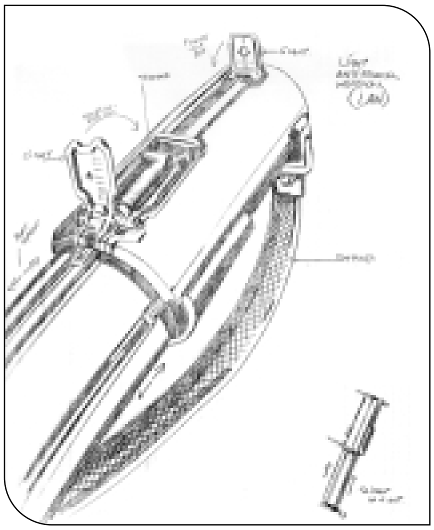


Guided Explosive Projectile (GEP) Gun

Ammo type: Rockets, WP Rockets
Base damage: 125
Clip size: 1 RDS
Rate of fire: SINGLE, 0.5 RDS/SEC
Reload time: 2.0 SEC
Recoil: LB
Base Accuracy: 75%
Acc. range: 900 FT
Max. range: 1500 FT
Mass: 50 LBS
Laser sight: N/A
Scope: YES
Silencer: N/A
Skill: Weapons: Heavy

The GEP gun is a relatively recent invention in the field of armaments: a portable, shoulder-mounted launcher that can fire rockets

Click on the Nanokey Ring to see a list of keys presently stored in the ring.



ABOVE LEFT. Everything in DEUS EX is about choices — who you are in the world, how do you interact in the world, what are you carrying, and so on. In this case, the player has clearly decided to go through the game as a heavy weapons specialist, despite the fact that this will leave little room in inventory for anything else. ABOVE RIGHT. One of the many weapons which can be chosen by players. RIGHT. A detailed weapon sketch.

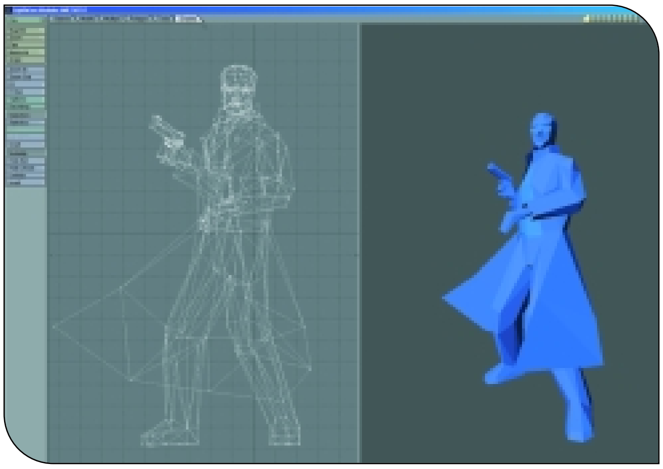
was something Harvey Smith had said over and over, from the day he signed on as lead designer. While some team members resisted this notion outright, I took a middle road, which just frustrated everyone. In the end, we cut a lot, left a lot, and made a game that everyone on the team was happy with (I think). This milestone made it clear that the time had come to make cuts, while giving us enough knowledge to cut intelligently. If we had waited until beta to make cuts, with just a few months to go before our ship date (as many developers do), it would have been a disaster.

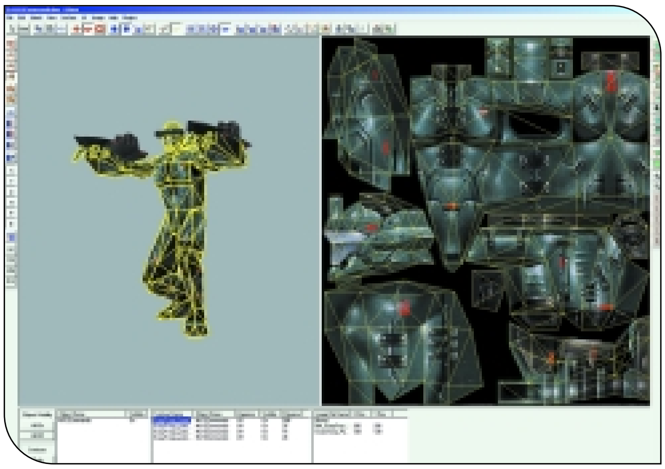
5. Licensing technology. We went into DEUS EX hoping that licensing an engine would allow us to focus on content generation and gameplay. For the most part, that proved to be the case. The UNREAL TOURNAMENT code we ended up going with provided a solid foundation upon which we were able to build relatively easily. Dropping in a conversation system, skill and augmentation systems, our inventory and other 2D interface screens, major AI changes, and so on could have been far more difficult. UnrealEd, the main tool our designers used to generate our maps, was superior to anything else available. UnrealScript was very powerful and allowed programmers to do lots of interesting things quickly and easily. The dollars and cents of the deal were right, and I didn't have to hire an army of programmers to create an engine, 80 percent of whose functions already existed in UNREAL TOURNAMENT. We were able to make what I hope is a state-of-the-art RPG-action-adventure-sim with only three slightly overworked programmers, which allowed us to carry larger design and art staffs than usual.

However, to my surprise, licensing technology didn't save us all the time I'd hoped it would. You'd think cutting a year or more of engine-creation off a schedule would result in an earlier release date. On DEUS EX, that didn't prove to be the case. Time that would have been lost creating tools was lost instead to learning the limitations and capabilities of "foreign" technology. Time that would have gone into making an engine went into focusing more on gameplay systems and tuning than normal. Unreal certainly allowed us to focus on content generation over everything else, but we spent more time doing it.

The biggest downside to licensing was that we were just never going to understand the code as well as we would have if we'd created it ourselves. That led to two distinct kinds of problems. First, there were areas where we ended up treating the engine as a black box. I think it's pretty well documented by now that we shipped DEUS EX with some Direct3D performance issues. Honestly, that didn't show up in any significant way during our QA process — a slight problem here or there, but none of the dramatic slowdowns some players reported in the early days following our ship date. Once players started reporting troubles, we were kind of in a lurch — we couldn't very well go in there and mess with the Unreal engine — we just didn't understand it well enough to do that safely. We had built around the edges of Unreal without ever getting too deeply into the nuts and bolts of it.

Second, because we didn't know the code inside out, and because we'd shelled out a fair amount of money for it, we tended to be conservative in our approach to modifying it. There were times when we should have ripped out certain parts of the UNREAL TOURNAMENT code and started





ABOVE. Bringing believable human characters to life is no easy task. The artists, whether working on concept art, 3D models, or texturing, had their work cut out for them. The job was made harder than necessary by a less-than-optimal team structure.

from scratch (AI, pathfinding, and sound propagation, for example). Instead, we built on the existing systems, on a base that was designed for an entirely different kind of game from what we were making. It's not that Unreal had bad AI or pathfinding or sound propagation, but those systems were designed for a straightforward shooter, which was not what we were making.

Technology licensing bought us a lot and cost us somewhat less. I guess the fact that we'll be licensing technology for our next round of projects, DEUS EX 2 and THIEF 3, says the price was right. But it remains an interesting dilemma, and we will be able to approach our next licensed engine with the wisdom gleaned from using Unreal for this project.

What Went Wrong

1. Our original team structure didn't work. You'd think after 17 years of making games and building teams to make games, I'd have a clue about team structures that work and those that don't. Ha! When I started pulling the DEUS EX team together I had a core of six guys from Looking Glass's Austin office. Having tapped Chris Norden to be lead programmer, I needed to find a lead designer and a lead artist. As I started casting about for the right person for the design job, something really good, but ultimately really bad, happened — two guys came along with enough experience to expect a leadership position. Instead of doing the sensible thing and picking one of them, even if that meant the other chose not to sign on, I got cute. I created two design teams, each with its own lead.

I put together two groups of people with differing philosophies — a traditional RPG group and an immersive sim group. We were making a game designed to bust through genre boundaries, and I thought a little competition and argumentation would lead to an interesting synthesis of ideas. I thought I could manage the tension between the groups and that the groups and the game would be stronger for it. My plan didn't work.

The design team was fragmented from the start. We had to name one of the groups "Design Team 1" and the other "Design Team A." (Neither group would settle for "2" or "B.") It became apparent — later than it should have — that I was going to have to merge the two groups and have a single lead designer. When I finally made that change I disappointed some folks, but the game was the better for it, and that's what's important in the end.

There were also challenges on the art side. DEUS EX suffered dramatically because for over a year, the artists "on the team" worked not for me or for the project, but for an art director in Ion Storm's Dallas office. Don't misunderstand — the art director was a talented guy. But talent doesn't make up for a matrix management structure (wherein resources in a department or pool are "lent out" to a project until they're not needed anymore) ill-suited to the game business, and it doesn't make up for being off-site. During this time, the art department drifted a bit. It was unclear whether the artists worked for me or for the art director in Dallas. I couldn't hire, fire, give raises to, promote, or demote anyone on the art team. We were assigned some artists who weren't interested in the kind of game we were making. The matrix management experiment made things a little tense, and presented many unanswerable dilemmas. Matrix management may work in some circumstances, at some companies, in some businesses. But I've never seen it work in gaming, and I've seen it attempted at three different companies. It especially doesn't work when one of the department managers isn't on-site.

I argued for a year that matrix management had failed at Origin and at Looking Glass. I had no doubt it would eventually fail at Ion. Eventually I got my way, and things got much better on the art front once the artists were officially part of the DEUS EX team. Still, I can only imagine how DEUS EX might have looked if we'd been one big happy team, including the artists, from the start.

If the experience of DEUS EX taught me one thing, it's the importance of team dynamics. You have to build a team of people who want to be making the game you're making. You have to deal with personnel issues sooner rather than later. And there has to be a clear chain of command. Many decisions can be made by consensus, but there can only be one boss for a project, there can only be one boss for each department, and department heads have to answer to the person heading up the project.

2. Clear goals are great . . . when they're realistic. We started out thinking very big. That in itself isn't bad — it's necessary to advance the state of the art — but we were unrealistic, blinded by promises of complete creative freedom, and by assurances that we would be left alone to make the game of our dreams. A really big budget, no external time constraints, and a marketing budget bigger than any of us had ever had before made us soft.

Let me give you some specific examples of ways in which we outreached ourselves in the original design of DEUS EX (before we made significant cuts). For one, there's no way, in a first-person RPG, to stage a raid on a POW camp to free 2,000 captives. Also, there's no way to re-create all of downtown Austin, Texas, with any degree of accuracy. Third, blinded by the power of UnrealScript, many of our original mission concepts depended upon special-case scripting and lots of it. We discovered the need for general solutions rather than special-case solutions later in the project than we should have (this despite much harping on the subject by some team members).

Find your focus early and maintain that focus throughout. General solutions are better than special casing. Give players a rich but limited tool set that can be used in a variety of ways, not a bunch of individual, unpredictable solutions to every problem. Always work within the limits of your technology rather than trying to make your technology do things it wasn't meant to do. Big budgets, lots of time, and freedom from creative constraints are seductive traps. Don't fall into them. Don't settle for less than greatness, but don't think too big. Balance should be the goal.

3. We didn't front-load all of our risks. In fact, we missed a big one. We were smart enough to realize we'd have to prototype and implement our new game systems early so we'd have time to tweak and refine them sufficiently. We did our conversation system and our complex 2D interface screens early, which was a good thing, too — they required as much tweaking as we feared. And in the end, they turned out pretty well, I think.

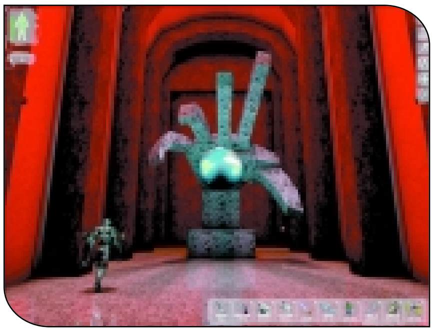
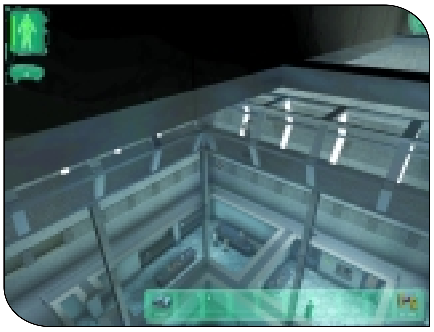
Unfortunately, we missed one huge risk area — artificial intelligence. I don't know how we missed it, but we did. It's not that we didn't spend time on AI. We started thinking about AI early in preproduction. Unfortunately, what that meant was that the AI was, to a great extent, designed in a vacuum, and as is often the case, we didn't really know what the game required with respect to AI until relatively late in development. And that meant implementing AI features early on that ended up being unnecessary later, once our design had evolved into its final form. In addition, building on the base of UNREAL TOURNAMENT's pure shooter AI meant that, instead of designing a system specifically for our needs, we ended up adding stuff and tweaking until the bitter end, causing NPC behavior to change constantly, right up to the last day of development.

We ended up with some pretty compelling AI, but the problem of convincing people they're interacting with real people is immense, particularly when you're talking about characters whose reactions have to run the gamut from fear to friendliness to violent enmity. That's not a challenge many games take on (with good reason), but it was one we had to take on for DEUS EX. Our sin was, I think, giving people a hint of what human AI could be in games, but delivering the goods inconsistently.

As I write this, we're discussing whether we want to risk making some fairly radical changes to the AI in a patch for a game that most people seem to like, and in which NPCs basically behave as much like real people as they ever have in any game. There's no telling which way our decision will go at this time.

4. Proto-missions redux. *Game Developer*'s Postmortems typically focus in on things the team clearly did right and things the team clearly did wrong. It sure is nice when things are that clear. Maybe it's just me, but I almost never see things in such black-and-white terms. Most of the time, problems are knotty and solutions are far from obvious or clear-cut, which is where the final two "What Went Wrongs" fall.

As I already mentioned, we recognized the need for proto-missions relatively early on, and built our schedule around the idea. We implemented two such missions, which helped us identify many things that didn't work (and many that did). With proto-missions in hand, we found ourselves at a critical juncture with two possible choices to make, the implications of which I still don't entirely understand.

ABOVE LEFT & RIGHT. Not all of the places players visit in DEUS EX were modeled after real-world spaces. The team had to make concessions to gameplay and create spaces more dramatic or more "3D" than one usually encounters in the real world.

On one hand, I could have gone off with some subset of the team and tweaked our proto-missions until they were absolutely right and models for all subsequent mission implementation before turning the rest of the team loose on implementation of the rest of the missions. On the other hand, I could have kept the entire team in implementation mode, getting all of the missions to the level of the proto-missions, meaning none of them would be exactly right but we'd be able to see the shape of the entire game and all of the missions would be ready for tuning at about the same time. The first approach would have left large portions of the team in thumb-twiddling or make-work mode for some unspecified period of time. This promised to prove that we could create a ground-breaking, compelling game, but could leave us without a finished game to ship. The second approach would have kept everyone productive throughout the project and at least put us in position to decide whether or not to ship the game at some foreseeable point in the future. The question was whether we would be able to turn all of the bare-bones missions into something fun or not.

I chose the latter approach and told everyone to get the game "finished" and playable at a bare-bones level. We'd worry about fleshing out all the missions, making the game as interesting and fun and dense and exciting as it needed to be during the inevitable gameplay tuning, tweaking, and balancing phase at the end. This probably isn't so much of a "What Went Wrong" as it is an open question of whether that was the right call. I think so, and the plan clearly worked to the extent that we shipped a game that people seem to like pretty well. But it's unclear to me whether using our proto-missions to fine-tune might not have resulted in an even better game.

5. Is it true that any publicity is good publicity? Naturally, this wouldn't be a complete or accurate picture of the development of DEUS EX if we didn't take a look at the Sturm und Drang that was Ion Storm. In case you you've been living under a rock, there's been a *lot* of hype surrounding the company. On the negative side, Ion Storm was heaped with bad press for much of 1998 and 1999. The company did the same things all game companies do, went through the same problems, but because we painted a big ol' "suck it down" target on our chests, the gaming press and a fair number of hardcore gamers went after us with a vengeance.

Not too surprisingly, this had an effect on those of us working away in the Austin office. Morale hits were frequent and problematic. It simply isn't possible to be bombarded by negative press about the company you work for and not take it somewhat personally. Trust me when I say that seeing your personal and private e-mails posted on the Internet is a devastating experience. Also, recruiting was more difficult than it should have been. We were able to put together an incredibly talented team for DEUS EX, but too many talented people told us that while they would like to work on DEUS EX, they couldn't work for Ion Storm. Eventually, a "we'll show them" mentality became prevalent in Austin. I don't know that anyone who worked on DEUS EX thought of him- or herself as part of the same company making DAIKATANA and ANACHRONOX up in Dallas. That kind of us-versus-them thinking is rarely good in the long run.

Now that we've shipped, the reviews seem to fall into two categories — those that begin with some statement implying that Warren Spector makes games all by himself (which is silly), and those that begin with some statement proclaiming that DEUS EX couldn't possibly have been made by Ion Storm (also silly). Silly or not, there's a level on which we're still trying to live down our past, at least in terms of the media's perception of our game and the company that paid the bills here.

But, for all the problems, being associated with Ion Storm wasn't all bad — far from it. On the plus side, it isn't as if anyone from *Rolling Stone*, *Entertainment Weekly*, the *New York Times*, the *L.A. Times*, *USA Today*, *Mother Jones*, the *Wall Street Journal*, *Forbes*, *Fortune*, *Time*, *Architectural Digest*, CNN, or the BBC ever banged down the doors at Origin or Looking Glass to talk to me or anyone on any of my teams. In reality, the bad publicity was almost entirely limited to the gaming press. The mainstream media, which barely notice anything about gaming (other than the fact that we supposedly turn normal kids into vicious killers) didn't seem to care about the bad stuff. But they sure did take notice of us. Ultimately, Ion's ability to attract attention to itself, even if it was sometimes in negative ways, probably worked to our advantage. Whether publicity at any cost is good or bad is still an open question for me.

The Bottom Line

Part of the challenge of game development is making the tough decisions along the way, the many difficult junctures when you have to determine that something that can't be done right in the game shouldn't be done at all. It's all well and good to have design goals and an ideal game pictured in your head when you start, but you have to be open to change and realistic about what can and can't be done in a reasonable time frame, for a reasonable amount of money, with the personnel and technology available to you. And if you don't have time to do something right, cut it and do everything that's left so well that no one notices the stuff that isn't there.

I'm not saying we did that perfectly on DEUS EX. We certainly didn't ship a perfect game. But if we hadn't gone into development with the attitude that we'd do things right or not at all, we would have fallen far shorter of perfection than we did. How close we did get is something all of you can decide for yourselves. All I know is we're going to get closer next time.

Return

The AI is in the Mail

It seems every new game has a blurb touting its artificial intelligence. The Game Developers Conference has more talks on AI every year. Developers are allocating more CPU cycles to AI. At least three companies offer AI tools to game developers. As I came home from the American Association for Artificial Intelligence's first symposium on computer games and AI in April 1999, the future seemed rosy for game AI.

I soon left the game company I was working for at the time, and began applying for other game AI positions. It would go something like this:

"How much experience do you have with 3D graphics programming in C++?" my interviewer would ask.

"Some, but I'm responding to your ad for an AI programmer."

"How about DirectX experience?"

"I have a DirectX book bag that I sometimes carry my AI texts in."

"Well, what sort of graphical APIs do you use?"

"Um, can we clarify what we mean by AI programmer?"

"You make things move around on the screen."

"I'd call that a 'graphics programmer.' Making creatures smart is very different from putting pixels on the screen."

"Oh, you write finite-state automata!"

"That would be a 'scripter.'"

"We only use finite-state automata, because we need to draw a million polygons a second. We tried fancier AI once, and it was a mess — slow, buggy, unmaintainable."

"Maybe if you found someone with an AI background . . ."

The interviewer interrupts, "Oh, we're still committed to improving our AI. We're very excited about the improved AI in our next sequel."

"Oh?"

"The new NPCs will have twice as many polygons! . . . Anyway, there's a problem with your résumé. It seems you've spent a lot of time in college."

illustration by Ben Fishman

"To study artificial intelligence."

"We don't want a lot of absent-minded ivory-tower types. And most of your work experience is outside the computer game industry, so that doesn't count. Say, do you know any 3D graphics programmers?"

O.K., so not every encounter went like that. Some companies knew what they were doing. But I did not make any of it up.

Talk about AI is still mostly lip service. Sure, graphics hardware has freed up some CPU cycles for AI, but an appreciation for its fundamental importance to gameplay has not entered developers' mind-sets. No one would release a 3D shooter today with the QUAKE 1 engine, but people are not embarrassed to release strategy games with WARCRAFT-era AI. We still see shooters with intentionally dumbed-down AI, on the theory that players get a bigger rush shooting a lot of dumb monsters than a few smart ones. Many developers still justify shoddy AI by saying that players are better at imagining reasons why random behavior is intelligent than programmers will ever be at making intelligent behavior. When developers discuss immersiveness, they mention high polygon counts, color depth, animation, physics, and good voice acting, but seldom AI. The typical developer's reaction to the word "AI" is not one of excitement about possibilities, but fear for their frame rate and unit count.

One problem is that AI is not integrated into the design process early enough. The top three things that visitors to my game AI survey web site (www.cse.buffalo.edu/~goetz/AI/API/gameai.html) want are all variations on pathfinding.

AI will never reach its potential as long as it is confined to a supporting role. What designers really need to ask is not, "How could I make my game better with AI?" but, "What game could I make with better AI?" Looking Glass modeled the effects of shadow and sound on NPCs' perceptions, and was able to create THIEF, arguably an entirely new type of game. Imagine what war games would be like if players could issue orders such as "secure this area," instead of building structures one by one. What if the computer tried to fool players

continued on page 63

continued from page 64

about where its forces are? What if units could command other units? What if they could disobey?

This leads to a second problem: legacy gameplay and game design tunnel-vision. We need to rethink gameplay paradigms that evolved to meet AI limitations that no longer exist. We need to have the courage to build games that players don't yet know they want. It may take a WOLFENSTEIN 3D of AI to change industry attitudes. If that happens, it's not going to be an AI retrofit of an existing game, but rather a design arrived at by thinking carefully about AI capabilities and their gameplay potential.

A final problem is that developers don't acknowledge that AI is a specialty. On a résumé, they value non-AI work within the game industry more than AI work outside the industry. They don't want an AI specialist; they want a person who is someone they would normally hire, "plus AI." The results end up being similar to what computer game art was like in the early 1980s, when it was assigned to whichever programmer had done the most doodling in high school.

As a result, buggy AI, slow AI, predictable AI, and inhumanly perfect AI have made developers more afraid of too much AI than too little. The director of a well-known massively multiplayer online game told me that they had "tried AI" unsuccessfully, and did not intend to try it again.

I had a martial arts instructor who would say "no" when asked if he knew karate, because he believed no one person could ever "know" karate. AI is like that. If you don't have enough work to hire a full-time AI specialist, you would do better to hire an AI consultant than to try to find a graphics programmer who also "knows AI."

The situation is better than it was a year ago. More resources are being spent on AI, but we can have great game AI now. All that game developers have to do is to want it. They have to want it badly enough to find out what it is, include it in the design process, and hire people who can do it.

PHIL GOETZ | *Phil has a doctorate in artificial intelligence from SUNY-Buffalo. He has been a roboticist, a mathematician, and a game AI programmer at Zoesis. He is currently working at Intelligent Automation Inc. on partly automated air traffic control. You can reach him at philgoetz@yahoo.com.*

ADVERTISER INDEX

COMPANY NAME	PAGE	COMPANY NAME	PAGE	COMPANY NAME	PAGE
3Dlabs	6	Discreet	19	Rainbow Studios	61
Alias|Wavefront	9	Macrovision	C3	Savannah College Of Art	62
American Inst. for Computer Sciences	21	Metrowerks Inc.	29	SGI	13
Anthony Mitchell	62	Multigen	14	Softimage	2
Apple Computer	C2,1	Musicandsfx.com	62	Sony Computer	59
Auran Development Pty Ltd.	60			Sound Werx	22
		Newtek	25	Staccato Systems	11
Cyan Inc.	60	Numerical Design Ltd.	5	Swingin' Ape Studios	60
Dice	60	RAD Game Tools	C4	Vancouver Film School	62