GAME DEVELOPER MAGAZINE

OCTOBER 2000

UBM Tech

Exit, Stage Left

Writing has never come easily to me. In fact, I still react to seeing my name in the magazine with some incredulity. I can still distinctly recall the joy I felt ten years ago as I was handed my college diploma, knowing that I would never have to write another paper. God throws some wicked curveballs.

Being the editor of *Game Developer* for most of its existence has been a wonderful experience, and yes, I've grown to love writing. But you know what they say about all good things, and it's time for me to move on. This is my last issue working on the magazine, but in reality, I'm not going far at all — in fact, I'm not even moving my office. I'm moving into a full-time producer position on *Game Developer*'s sister site, Gamasutra. The siren song of the online world beckons!

I've had an incredible time working on *Game Developer*. It doesn't seem that long ago that a bunch of us on the staff of *Software Development* decided to secretly siphon off money from our magazine to launch *Game Developer*. It was daring stuff for us — we weren't sure how our superiors would react to the fact that investment intended for one magazine was actually going into the unapproved launch of another. Thankfully the gamble paid off: the reader response to our initial issues was strong, and that gave us enough courage to break the news about our guerrilla launch to executives at our company. Instead of firing all of us, they kicked in official investment, and since then we've tried to improve the quality of the magazine every month. That improvement is due in large part to the feedback and contributions of people in the industry like yourself.

There are so many people who have helped shape the magazine into what it is today, and I'd like to thank all of them by name. But I'd easily burn the rest of my word count on this column trying to do so. So I've narrowed it down to two developers who have been extremely supportive from the beginning of my time on the magazine.

First, I'd like to thank Hal Barwood of LucasArts. Every month, for years now, Hal has sent me a long e-mail containing thorough feedback about every article, column, and review in the latest issue. The staggering amount of constructive criticism Hal provided has improved the magazine in many fundamental ways over the years. One of my biggest challenges as editor was to get him to write for the magazine, and one of my biggest regrets was that I never succeeded. Hal, I hope you're saving up all that wisdom for a book.

The second person who's been great to me is Chris Hecker. It's pretty amazing how much time and energy Chris devotes to bettering this industry, and he's been a constant source of information and inspiration. Chris is a true asset to the entire industry.

Welcome Mark, Adios Alex. In my column over the years I've introduced and bid farewell to quite a few magazine contributors, and it's my final privilege to introduce the new captain of this ship. Mark DeLoura joins us from Nintendo, where he was the software engineering lead in charge of third-party titles, and worked on one of my all-time favorite games, LEGEND OF ZELDA: OCARINA OF TIME. You also might recall the article he wrote for the magazine last November ("Putting Curved Surfaces to Work on the Nintendo 64"), or maybe you've seen his brand-new book, *Game Programming Gems*. Mark brings a wealth of experience and expertise to this magazine, and besides all of that, he's one of the nicest people I've ever met in the industry. It's good to know that the magazine's best days lie ahead with Mark at the helm. Welcome Mark, and to all you readers, thanks for the memories.

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

Editorial Director
Alex Dunne *adunne@sirius.com*
Editor-In-Chief
Mark DeLoura *mdeloura@cmp.com*
Senior Editor
Jennifer Olsen *jolsen@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 Susan McDonald
Product Marketing Manager Darrielle Sadle
Field Marketing Manager Jennifer McLean
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

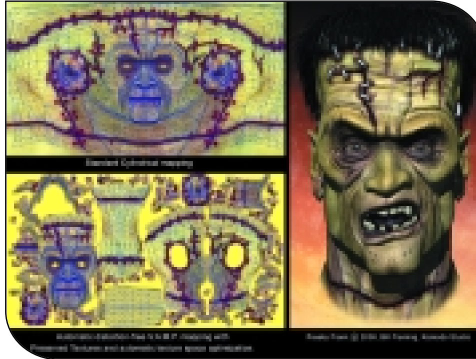
WWW.CMPGAME.COM

FRONT LINE TOOLS

WHAT'S NEW IN THE WORLD OF GAME DEVELOPMENT | *daniel huebner*

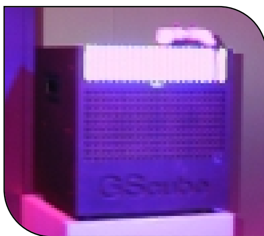
NEW TEXTURING ARSENAL FOR DEEP PAINTERS

Right Hemisphere is shipping Texture Weapons, an add-on for its Deep Paint 3D texturing and painting program. Texture Weapons is designed to extend Deep Paint 3D with enhanced painting and mapping functionality, and to let users create textures with minimal distortion. Texture Weapons includes automatic and manual UV editing tools as well as Projection Paint, a feature that allows users to paint from any position within a 3D space with brush and texture size unaffected by UV-coordinate variations. Texture Weapons is available now, and is priced at $495 for Windows 98 and NT.



Texture Weapons | Right Hemisphere | www.righthemisphere.com

SONY UNVEILS PLAYSTATION 2—BASED RENDERING SYSTEM

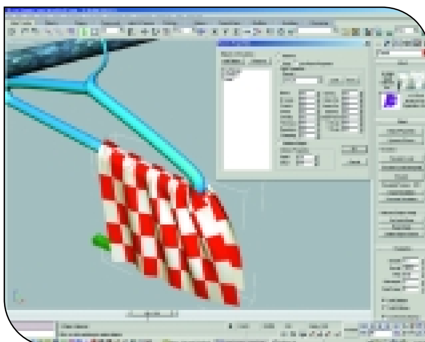


Sony Computer Entertainment has rolled its Playstation 2 development research into a new rendering system. The new machine, called the GS Cube, boasts 16 parallel graphics units, each one composed of the PS2's Emotion Engine CPU and an enhanced Graphics Synthesizer GPU bumped up to 32MB of VRAM from the PS2's 4MB. Sony envisions a truly interactive cinema experience, but concedes they are unsure of exactly what might be created with the GS Cube. Sony hasn't set a date for release or any pricing strategy, but could have GS Cubes ready for commercial use by the end of the year.

GS Cube | Sony Computer Entertainment | www.sony.com

NEW MAX PLUG-IN GOES FROM RAGS TO STITCHES

Digimation has a new plug-in for 3D Studio Max called Stitch, which aims to deliver a realistic cloth simulation complete with wind, gravity, and collisions. Stitch works as a modifier in Max, allowing users to select an object and apply the Stitch modifier to transform an object into cloth with all of its associated properties. Users can also identify surrounding objects in a scene to act as collision objects, and can add 3D Studio Max standard Wind and Gravity space warps to the simulation for added realism. Stitch comes with a library of preset cloth types and generic patterns, and costs $695 per license.



Stitch | Digimation | www.digimation.com

ELSA DEBUTS GLORIA III

ELSA is releasing the newest member of its Gloria line of workstation graphics cards, the ELSA Gloria III. Based on Nvidia's new Quadro 2 Pro, Gloria III offers nearly twice the processing power of the Quadro-based Gloria II. The new Gloria pushes up to 31 million triangles through its transform and lighting engine and fills textures at a rate of one gigapixel per second. Its four pixel pipelines each handle seven operations with two textures per pixel, per pipeline, per clock. With 64MB of DDR frame buffer memory and a 350MHz RAMDAC, the Gloria III will support up to 50-inch CRT monitors and 21-inch DVI flat-screen monitors with resolutions as high as 2048×1536 at 85Hz in true color.

Gloria III | ELSA | www.elsa.com

3D STUDIO gMAX

Discreet is extending its 3D Studio Max tools with an authoring tool available to game players at no charge. 3D Studio gMax, a professional and consumer content creation platform based on the next 3D Studio Max release, is aimed at developers who plan to release custom level-editing tools with game titles next year. Licensed game development teams

discreet™

will be able to use and extend gMax as a game level-building tool for internal production of a specific project, and then release those customizations as a Game Pack that users can plug into a free, downloadable gMax editor from Discreet. The new product will be compile-compatible with the next major release of 3D Studio Max, and Discreet plans to have the no-charge download of 3D Studio gMax available to consumers when the first set of supported Game Packs are released.

3D Studio gMax | Discreet | www.discreet.com

Sony prepares for U.S., European PS2 launches. Sony Computer Entertainment president Ken Kutaragi announced that 3 million Playstation 2s have shipped in Japan as of August 1, less than five months after its March 4 launch. By comparison, the original Playstation took 19 months to achieve the same feat, having moved more than 77 million units worldwide to date. Several leading U.S. retailers have halted or scaled back preorders for the new console, anticipating that supply will not be able to match demand at the October 26 launch. Some of the retailers are still taking preorders through their online stores, while others have eliminated guarantees that preordered consoles will be shipped on the launch date.

European players, meanwhile, will have their patience tested by a delay. The Playstation 2 will arrive in 17 European countries November 24. That date is nearly a month after the U.S. launch on October 26, and even later than the previously expected release date of November 17. Sony Computer Entertainment said the first wave of Playstation 2s shipped to Europe will still out-pace the 600,000 units it shipped in the first three months of the original Playstation's European launch. Sony has also set its PS2 pricing for Europe: 299 pounds ($447) in the U.K., 2,990 francs ($413) in France, and 869 marks ($403) in Germany.

Microsoft flexes marketing muscles. Microsoft plans to spend a cool half-billion dollars on marketing its Xbox in the first 18 months of its debut, making it the biggest product launch ever for Microsoft. The plan at this point appears aimed primarily at unseating the prominence of Playstation 2, which will have been on the market about a year before Xbox launches in fall 2001, and staving off any threat from Nintendo's next-generation console, which is expected to arrive in the U.S. at around the same time as Xbox. A presentation Microsoft gave to a meeting of financial analysts was focused mainly on comparisons between Xbox and Playstation 2 specifications and development environments, leading some analysts to believe that Sony is the only competition Microsoft is taking seriously. However, much less is known at this stage about Nintendo's upcoming console than Playstation 2, which has already been released in Japan.

SOLDIER OF FORTUNE: too gory for some Canadians.

The company took another big step in the development of the Xbox with the shipment of the first Xbox development kits. The company is refraining from calling the kits SDKs, preferring the more clever name "XDK." The company is expecting to ship more than 1,000 kits to some 100 development houses in preparation for the console's launch. The XDK is being rolled out in three phases, starting with an upgradeable PC-based system and finishing with a customized console unit delivered in the first part of 2001. The kits currently shipping provide developers with early graphics hardware, a beta version of DirectX 8, and off-the-shelf game pads to simulate Xbox functionality.

Interplay stays listed. Interplay will continue to trade its stock on the NASDAQ's National Market. The company had faced losing its National Market listing because of its failure to meet all of the market's listing requirements, but an appeal to the Listing Qualifications Panel came down in Interplay's favor. The panel decided that Interplay has demonstrated compliance with NASDAQ's net tangible assets criterion and may keep its listing as long as the company's assets do not fall below $8.3 million in its third quarter. Interplay must also provide documentation of compliance with NASDAQ's voting requirements.

Another company facing delisting, Acclaim, lost it appeal to continue trading on the National Market, where it also failed to meet the exchange's net tangible assets listing criterion, and will begin trading on NASDAQ's Small Cap Market. Acclaim has been hit hard in recent quarters by the console transitions, and is currently in the process of reworking its product line in an effort to return to profitability.

Too Grizzly for B.C. Authorities in British Columbia have given Raven's SOLDIER OF FORTUNE the equivalent of an X rating. The game is the first in Canada to be classified as an adult movie, making it illegal to sell the title to anyone under 18. The classification, which carries penalties of up to six months in jail and a fine, came after a parent complained about the game's violent content. SOLDIER OF FORTUNE publisher Activision announced that it will appeal the decision, and the game's Canadian distributor, Beamscope Canada, has also filed a similar appeal. Both companies contend that the B.C. Film Commission doesn't have the jurisdiction to rate and classify games. The action has led the province to develop its own mandatory game ratings scheme, similar to that used to rate movie content.

Take-Two grabs Pop Top. Take-Two Interactive is buying Pop Top Software, best known for developing last year's hit RAILROAD TYCOON 2. In addition, the sale includes a provision to keep Pop Top president Phil Steinmeyer with the company in his current capacity for at least four more years. The terms of the sale have not been disclosed. Pop Top is currently working on TROPICO, a 3D strategy game set for release in 2001, and will remain in its St. Louis headquarters.

UPCOMING EVENTS CALENDAR

SOFTWARE DEVELOPMENT EAST
Washington Convention Center
Washington, D.C.
October 29–November 2, 2000
Cost: full conference starts at $1,695
www.sdshow.com

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

Timestep Regulator

a.k.a. Framerate Regulator, Real-Time Timestep, Fixed Timestep

Problem

Many games strive to achieve "real-time" behavior, meaning the game appears to behave the same regardless of machine speed and computational load. The in-game speed of objects should not be tied to the framerate; if an object is supposed to be moving at five units per second, it should move at five units per second on all machines under all circumstances, so all users experience the same gameplay.

Older games that violate this principle often run too fast to be playable on new machines. Similarly, the gameplay on modern games is sometimes too slow to work as intended on even slightly older machines. A more subtle manifestation of this problem occurs when the gameplay changes speed due to fluctuating in-game processing overhead.

Solutions

Fixed Timestep. If a suitable target machine speed and computational load can be determined a priori, and the framerate can be kept relatively constant, the Timestep Regulator can be a simple constant fixed timestep. Alternatively, a set of fixed timesteps can be used within precomputed zones or situations.

Variable Timestep. If the frame time varies unpredictably, a variable timestep can be used as the Timestep Regulator to keep the "game time" approximately synchronized with "real time." The variable timestep is usually based on the previous frame's time, or an average of a history of frame times, to filter spikes in the framerate. More advanced implementations can keep track of different subsystem times to predict future frame times more accurately and help avoid the "Evil Feedback Loop" issue mentioned below.

Issues

Fixed Timestep issues. If the actual machine running the game does not match the target machine speed, the game will not have real-time behavior. Many PC games exhibit a bias toward high-performance machines for this reason. In extreme cases, such as when a game developed for a slow machine runs on a fast machine, a framerate limiter may be employed to keep the game from running too fast.

Miscellaneous Variable Timestep issues. Initialization of the history is problematic, and it's possible to overshoot the current real time due to spikes. There is debate over whether games should simulate up to the beginning of the frame, or to the exact moment the user will see the updated image after rendering.

Evil Feedback Loop. This is the biggest danger for Variable Timestep algorithms. Most advanced rendering, simulation, and AI algorithms use coherency — similarity between frames — to improve performance; the less coherency, the longer the algorithm takes to execute. For example, in a continuous level-of-detail terrain system that caches tessellations, the farther the eye moves, the more work the algorithm must do to generate the new ground polygons. A Variable Timestep can turn this into a destructive positive feedback loop: If a given frame takes a long time, the simulation will try to make up for lost time during the next frame and step forward by a large amount. This large timestep causes objects to move farther than usual, destroying coherence, and causing the coherence-sensitive algorithms to take even longer to complete than the previous frame. Once this loop begins, the frame times increase until an equilibrium is reached where the timestep and the run time match. This steady-state frame time is usually far too slow for interactive games. One solution is to detect the emergence of the Evil Feedback Loop and abandon absolute real time, stepping as far as possible within the simulation time budget.

Timestep linearity. The relationship between an algorithm's run time and its step size can be constant, linear, or nonlinear. For example, some simple AI algorithms do the same amount of work regardless of the step size. However, a collision detection system that prevents objects from tunneling through one another will do more work the farther the objects move. This relationship is important to understand in order to implement a Variable Timestep that predicts frame times accurately and avoids the Evil Feedback Loop.

Multiplayer synchronization. Keeping networked games synchronized is an issue with both solutions. This is a huge issue and will be the topic of future patterns. Be sure to submit your thoughts on this topic!

Uses, Credits, and References

Jon Blow from Bolt Action Software coined the term Evil Feedback Loop and used it in his Siggraph 2000 talk, but many developers have run up against the issue in many different games. Chris Hecker's articles on www.d6.com/users/checker/dynamics.htm show some simple Variable Timestep implementations.

We Want to Hear from You!

This column depends on your contributions! Send your patterns and idioms to us at patterns@d6.com. To learn more about this column and the Game Programming Patterns Database, go to www.gamasutra.com/patterns. If we publish your pattern in the column, we'll give you recognition in print and $100!

Physics Engines
Part Two: The Rest of the Story

by jeff lander & chris hecker

Last month ("Physics Engines, Part 1: The Stress Tests," September 2000), we put three licensable rigid body physics engines through their paces with a series of stress tests specifically designed to find stability problems. MathEngine's Dynamics Toolkit 2.0 and Collision Toolkit 1.0, the Havok GDK from Havok, and Ipion's Virtual Physics SDK all made their way through the 12 tests with varying levels of success.

This month, we are going to take a broader look at each package. But first, you are probably asking yourself...

Should I License a Physics Engine?

Game physics is a complex issue. We are not going to answer the question, "Should I use rigid body physics in my game?" That's a decision you will have to make. However, if (or, we hope, when) you decide to try physics in your game, should you build or buy? That too, is a complicated question.

Rigid body dynamics is a relatively new technology for the game industry. It is mathematically intense and difficult to implement. This would seem to make it perfect for licensing, because a third party has already done the hard part, and you can just buy the technology from them. This is true to some extent, but it remains to be seen how knowledgeable programmers and designers need to be about physics to use a physics engine. As you saw last month, physics is not a technology you can put in your game and have it work perfectly every time. Current simulators are quite sensitive to initial parameters and tolerances, and require hand-holding and tuning.

We feel that developers using physics simulators to make games in the near future will need to know a fair amount about physics and the underlying simulation algorithms in order for such implementations to be successful. However, we also feel this level of knowledge is less than what's required to implement a physics simulator in the first place, so licensing might be a good idea. You won't know exactly why things are breaking, but you'll be a year ahead of the guy down the street trying to implement his simulator from scratch.

A car built by connecting rigid objects with springs in Ipion's Virtual Physics SDK.

Criteria & Disclaimers

Even after reading this review, anyone who is contemplating licensing a physics engine should arrange for an evaluation period to determine how it might fit into an individual project. With a high-risk piece of core technology, there is no substitute for firsthand experience.

We reviewed each package as if we were going to license the package and either integrate it into an existing game or start a new game with the engine. These are subtly different goals, and we'll point out areas where this distinction is important later on. For this review, we looked at the following factors:

Feature set. What are the elements of each system? Can it be used to simulate a great variety of things, including machines, environments, and so on? Is the collision detector full-featured? How many constraint types are there? How many types of friction? We're focusing on the rigid body dynamics features, not things like cloth, soft bodies, water, or domain-specific subsystems such as cars and airplanes.

Documentation. Is the information included adequate and well presented? Does it address nonintuitive problems that arise? Does it discuss tuning tolerance parameters in detail? Are there clear and instructional sample programs included?

Ease of use. Are the libraries intuitive to program with?

Production. Does it include export and preview tools to simplify production and artist use? Can you save internal object representations on the disk so you don't have to let the engine preprocess them every time you load? Does the library include source code? Is it cross-platform?

Integration. Is it easy to integrate into a production? Is it modular, and are the modules well designed? Do the modules work well together?

Input and feedback. Can the game interact with the objects effectively via forces and impulses? Is useful information available from the simulation, such as object speed, contact state, and so on?

Cost. What is the cost, both in licensing and support?

Technical support. How well do the companies respond to problems?

What We're Missing

The most glaringly omitted criteria are stability and performance. We covered stability last month with the stress tests, and as we said in that article, we cannot cover all cases. You have to test situations specific to your game design.

THE AUTHORS | *Jeff Lander (jeffl@darwin3d.com) is the Graphic Content columnist for* Game Developer. *Chris Hecker (checker@d6.com) is* Game Developer's *editor-at-large.*

Performance is another complicated and hard-to-review feature. All three engines seem fast in the demos, but we implore you to try them on your representative data sets before making a judgment. Related to performance are memory and resource usage. Does one engine allocate 100MB under certain special circumstances? We don't know.

We haven't covered track records or postmortems. To our knowledge, no hit games have shipped with any of these SDKs. We've talked to a few developers using the engines, but a thorough set of interviews and developer feedback would be very interesting and useful.

Before we get into the reviews, we must note that Ipion was recently purchased by Havok, and the Virtual Physics SDK will eventually be integrated into the Havok GDK. For the time being, we will treat them as separate products and consider them separately here. The packages are presented in no particular order.

Ipion's Virtual Physics SDK

Features. Ipion's collision detection system has all the necessary features, including broad and narrow phase algorithms, and functions and utilities for generating collision volumes from meshes. You can apply forces to the objects through actuators, such as motors, springs, and even a simple buoyancy simulation. The system supports a constraint system where you can specify the degrees of freedom between the constrained objects or use built-in joints, such as hinge or ball-and-socket.

Documentation. The manual is a relatively scant 76-page Microsoft Word document. It covers the main features, but it does not go into great depth, making thorough examination of the samples and header files imperative. It would have been very helpful to have all of the classes fully explained and an overall description of the architecture. Tuning parameters are hardly documented. The examples are numerous, providing a demonstration of the complete functionality of the system, as well as creating a starting point for most situations one would typically encounter in the development of a game. The example code comments are a little sparse. All of the examples are in an example framework, so it's hard to tell how to use the engine on its own.

Ease of use. The library has been engineered as a class hierarchy. Objects need to be allocated and owned by the system. Sometimes the public sections of classes have many "comment enforced" rules. We could not find a clear way to control the simulator step-size. No access is provided to the low-level dynamics algorithms, such as the contact solver.

Production. Ipion doesn't include much in the way of production tools. There are no exporters for standard 3D modeling packages, nor any importers for generic 3D model files. QUAKE 2 .BSP files are supported with sample code. Most of the example programs generate models using custom code. Users would need to create tools in order to let artists try out their models easily. Ipion does provide a tool for creating a custom convex hull for complicated concave models; however, this tool doesn't load any standard file formats, either. There is an example of saving internal collision information to disk, but it is not documented and doesn't appear to be cross-platform-safe or version-controlled. The engine supports PC and Playstation 2. Ipion has licensed the complete source code to game developers in the past, although it's not clear if the Havok acquisition will affect this stance.

Integration. As mentioned above, the class hierarchy might make integration with an existing architecture more difficult than with a more procedural API. The classes are well organized and mostly documented. The libraries are composed of two pieces, the physics engine and the surface builder, which allows users to create optimized convex hulls from polygon meshes at run time. If you don't need the surface builder function, you can choose not to include it.

Input and feedback. The engine has a few different ways of physically affecting the simulation, including applying impulses to objects and attaching actuators and controllers (like springs, magnets, and user-defined impulses). Ipion includes a class of "listeners" that allow the game to get information about objects and collisions in the system. With these you can determine if objects are actively being simulated or are "asleep." The collision events are very important for adding things like sound effects to a game. These listeners look very useful, though they are not terribly well documented.

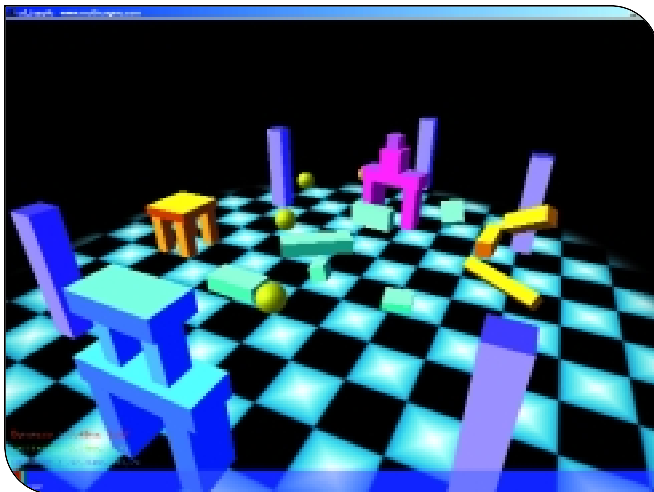
Cost. The base price for the Ipion SDK is $50,000 to $60,000 for a single title. Royalty-based fees can be negotiated on an individual basis.

Technical support. Technical support for the Ipion system is now handled by Havok. They have a web-based forum for answering developer questions as well as telephone and e-mail support. It is not yet clear how support will be handled between the Havok and Ipion packages.

MathEngine's Dynamics Toolkit 2.0 and Collision Toolkit 1.0

Features. The MathEngine system is composed of two pieces intended to work together. The Dynamics Toolkit simulates rigid bodies, constraints, and contacts. A number of constraints and contact friction types are included, and access to the dynamic simulator is provided at a number of levels. The Collision Toolkit handles collision detection between objects, but it is very basic. It only sup-

Collision detection with random polygons in Ipion's Virtual Physics SDK.



A dynamically controlled object in a scene in MathEngine.

ports a small set of collision primitives (sphere, box, plane, and cylinder) and their unions. The API is modular (more on this below), so adding a custom collision detector is possible, but it would be a lot of work — work you were hoping to save by licensing a physics engine.

Documentation. MathEngine's documentation is the best of the three packages. The manual is very thorough with both a user's guide and complete reference to all the classes in the architecture. There is even a glossy foldout that shows frequently used information. The code is well organized and documented. There is a small amount of inconsistency in the documentation; for example, it uses the terms alive/awake and dead/asleep interchangeably. The friction parameters are nonintuitive and nonphysical. Some guidelines and a bit more documentation on setting them would have helped. The tolerance values used by the simulator are documented, and some effort is made to describe them. Additionally, there is a section on optimizing and constructing good simulations. While easy to read and understand, the demonstration programs were too basic.

Ease of use. The system is a class hierarchy in structure, but the access to the library is through fairly lightweight standard C functions and data structures. The API is clean and relatively well designed. The dynamics and the collision toolkits are completely modularized and loosely coupled, so a developer could easily use either the toolkit by itself, or call the well-documented contact-resolver function directly. This layered approach is the right way to design a physics engine API for licensing, in our opinion. However, MathEngine has not provided a complete interface between the systems, so you have to do extra work to get them to cooperate. The system has no matrix helper functions. Simulation memory allocation is error-prone.

Production. Similar to the Ipion library, the MathEngine system is designed mainly for coders. Any tools needed for importing models or allowing artists to preview the system will need to be created. The Collision Toolkit can use Renderware models, but no other formats. There appears to be no way to write internal preprocessed data to the disk. The engine supports the PC and Playstation 2 platforms. MathEngine does not generally license complete source code, but said they would discuss it on a case-by-case basis.

Integration. The layered approach to the API makes integration very easy, probably much easier than the other two engines. If your program already handles collision detection, you can easily use only the dynamics system or vice versa. MathEngine provides most of the source code to the abstraction layer that communicates with the dynamics and constraint solver.

Input and feedback. There are simple APIs for exerting torques and forces on bodies, but the programmer must apply joint forces and torques manually. Some of the toolkits and subsystems have callback functions for feedback, while others have "get" functions; most of the rest have enough sample code included that direct access of the data structures is possible.

Cost. MathEngine has two pricing plans, either a single fee of $50,000 per project, or $5,000 plus royalty of 50 cents per unit.

Technical support. The MathEngine web site has a developers' forum for discussing the toolkits. Support over both e-mail and telephone is very good, although we did not test this support anonymously.

Havok GDK 1.2

Features. Havok provides an underlying rigid body simulator and a toolkit for higher-level access layered on top. Havok's constraint system is weak compared to the other libraries, supporting only spherical joints and point-to-path. The collision subsystem is full-featured, supporting simple collision volumes as well as convex hulls and true concave objects. A number of contact-solver friction types are included. In addition to basic rigid body dynamics, the system simulates soft body objects (such as blobs and cloth) and particles, as well as a simple fluid model; however, we didn't test these features and can't vouch for their completeness.

Documentation. Havok provides a nice set of documentation along with updates on Havok's web site. While not as thor-

OTHER CHOICES

There are other options beyond the rigid body dynamics simulators we discussed here. Several middleware providers have created tools to aid the development of physical simulations. Here are a couple we haven't looked at closely, so "caveat programmer."

The ReelMotion Animation Tool (www.reelmotion.com)is a simulator for generating animation data for a variety of vehicles such as cars, helicopters, airplanes, and motorcycles. It uses dynamic simulation and sophisticated physical models for the objects to generate the motion data.

Hypermatter from Second Nature (www.hypermatter.net) provides a real-time system for animating soft body objects. The stiffness of the object can be controlled so the objects can vary from rigid to very soft while still preserving the volume of the original object. The system contains a number of features common to a rigid body simulator, such as constraints. The toolkit is currently available for the PC, and a Playstation 2 version is in development.

Dynamically controlled objects in a scene in Havok.

ough as MathEngine's documentation, the documents were useful. There were a few minor consistency problems with function names and terms. There is a great variety of well-documented examples of various levels of complexity. The underlying simulator API is not as well documented.

Ease of use. The Havok GDK toolkit is very easy to use, with the code organized in an easy to understand class hierarchy. The system provides helper utilities for common math functions. The underlying simulation API seems slightly more difficult to use and not quite as completely exposed as MathEngine's, but is more complete because of the advanced collision detector.

Production. Because the system provides a plug-in for 3D Studio Max, artists can easily start using it. They can create models with boundary volumes and run them in the simulation without programmer involvement. Another nice feature allows you to dump the state of the objects in the simulation at any point to a file to reload later or examine for debugging purposes. When trying to debug a physics-heavy game, features like this are very useful. The library is available for the PC, Macintosh, and Playstation 2 platforms. Havok does not license the engine source code, though they will discuss full source-level needs individually.

Integration. Havok breaks the package up into several libraries. While it may be possible to leave out unused portions, they are very tightly integrated. This makes programming easier at the expense of modularity.

Input and feedback. Input to the simulation is through "action" classes that are called back during integration, and through a complete set of access functions on the bodies. Because the constraint system only supports one type of constraint, there are no constraint actuator functions. The Havok GDK provides a complete set of event callbacks and access functions to determine what is happening inside the simulator.

Cost. The Havok 3D Studio Max plug-in is $495 per seat with multi-seat discounts available. The Havok GDK is available for $65,000 to $75,000 per title without royalties. Royalty pricing and other pricing options are available on an individual basis.

Technical support. Like MathEngine, the Havok web site has a developers' forum for discussion of the toolkits. Technical support was very good, but again, it was not attained anonymously.

Conclusions

All three of these packages performed better than we expected. They are clearly up to the task of handling most rigid body simulation needs. But which one is right for you? We arrived at some general conclusions.

While its collision detection system is very fast and flexible, the Ipion Virtual Physics SDK has some problems. It lacks adequate documentation, and the code architecture may make integration with your game application more difficult. It is a product at the end of its lifetime, and unless your needs are very near-term and it is a perfect fit for your code structure, we would hesitate to recommend it. We do hope that Havok can integrate many of Ipion's good features into their architecture.

The Havok GDK is the most full-featured physics engine available. The system handles collision detection of arbitrary geometries and has models for soft bodies, particle systems, and basic fluid dynamics. The documentation is fairly good (though we hope it continues to improve), and the code examples are plentiful and very useful. The availability of a 3D Studio Max tool for artists is a great aid to production. If you need a one-stop, do-everything physics engine, this is the one for you. The only weak points are its constraints (admittedly, a very large weak point, especially if you're interested in simulating articulated figures), library modularity, and occasional collision resolution issues.

The MathEngine Dynamics Toolkit 2.0 and Collision Toolkit 1.0 is a very well designed SDK. The documentation is the best of the three, and technical support is outstanding. The well-documented architecture makes the engine easy to integrate with existing game projects. Programmers with knowledge of dynamic simulators should be able to dig right down into the core and control the simulation. The dynamic simulator is very good, and the rigid-constraint support is the best of the three systems we looked at. However, the lack of advanced collision detection methods might be a showstopper of a problem. (We must note here that as we went to press, we learned that a new beta version of the MathEngine Toolkit was released that includes collisions with convex objects among other new features. We didn't have time to check this out thoroughly, but because all of these packages are constantly evolving, we again encourage you to do your homework when weighing your options.) Finally, the simplicity of the demos may also make it tough to find specific examples to build on.

As we've said many times, you must review the packages for your specific needs before making a decision. Physics is not mature enough to buy a package based simply on a recommendation. All three packages are available for evaluation, and this is really the only way to make sure you get the package that is right for you and your project.

FOR MORE INFORMATION

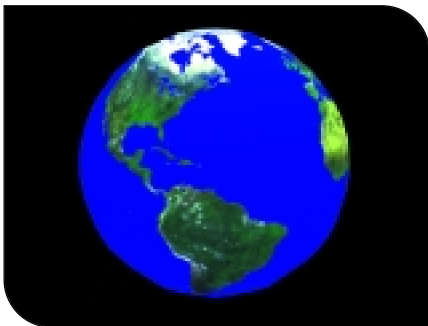
Havok GDK and Ipion's Virtual Physics SDK
www.havok.com

MathEngine's Dynamics Toolkit 2.0
and Collision Toolkit 1.0
www.mathengine.com

That's a Wrap

Texture Mapping Methods

Many artists and programmers struggle with the issue of applying a flat texture to 3D objects. There are many choices to make. The artist must decide which type of mapping will provide the most complete coverage with the least distortion. Aligning or attempting to hide the texture seam is also a pretty complicated task. It can often be torture. However, mapping is not a new problem.



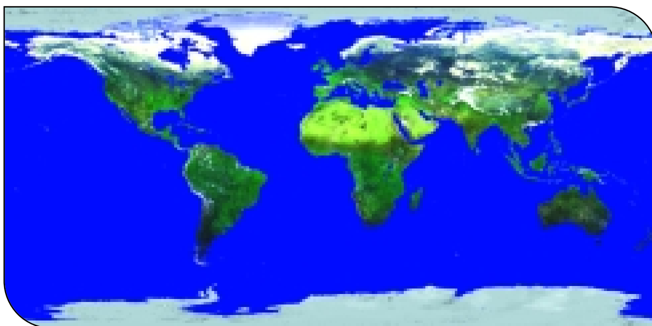


FIGURE 1. The earth (top) and its Mercator projection (bottom).

In the 16th century, a German cartographer named Gerhard Kremer was worried about spherical mapping. His goal was to unwrap the globe onto a 2D map for use by ocean navigators. Up to that time, maps of the world did not take into account the curvature of the earth. In order to navigate a ship from one port to another using such a map, you could not simply draw a straight line between the ports and follow that course. Instead, you had to adjust your heading along the way to compensate for the earth's curvature. Gerhard, who, in the educated fashion of the day, Latinized his name to Gerardus Mercator (Kremer is an old German word meaning "merchant"), studied the problem of map navigation. Sailors at the time were able to guide their boats at a constant heading fairly easily; therefore, they needed a map where a straight line defined a constant heading. The concept of lines of longitude and latitude were well understood by this time. Mercator created a map where both the lines of latitude and longitude were parallel, straight lines. This map projection is now known as the Mercator projection, or more formally, a cylindrical, orthomorphic, conformal projection. While we can easily create this projection with a computer, Mercator created this map with just a compass and protractor.

This projection made traveling easier for sailors, but it had a few drawbacks. Although sailors could now travel between two destinations on a constant course, because of the curvature of the earth this was not always the shortest distance between two points. To this day, measuring distance on a Mercator map is not entirely accurate. You may have noticed this if you have ever traveled to Europe over the polar route — on the map it looks like you are not traveling the shortest route. Also, in order to keep the longitudinal lines from getting closer at the poles, the scale must be increased with distance from the equator. This leads to quite a bit of distortion near the poles as well. Many people assume that Greenland is the size of South America from looking at maps, when it is actually quite a bit smaller.

Despite its limitations, the Mercator projection remains an incredible achievement, and is still in common use today. Unfortunately, like many great thinkers of his day, as a reward for his brilliance and his Protestant beliefs, Mercator became a guest of the Inquisition for almost a year.

Mapping and Me

Fortunately, mapping today doesn't usually mean actual torture, but it is still quite a pain. In real-time 3D art, we are usually concerned with the opposite problem that Mercator had — given a 2D map, we would like to apply it to a 3D object. The goals are pretty much the same, though: make the map as accurate and free of distortion as possible. Artists continually struggle with the annoying artifacts that various mapping methods create.

Last month, I discussed a 3D paint application ("Art and Intelligence: 3D Painting," September 2000). To have a canvas to paint on, I needed a texture applied to the object with texture coordinates at each vertex. At the time, I relied on objects that already had mapping coordinates from a 3D modeling package. Nevertheless, many times I want to start with an object that has never had a texture applied and start painting on it. That means I need to generate the texture-map coordinates myself. This is where I turn back to mapping projections. This month, I'm going to explain some basic texture projection concepts and build up a toolkit of texturing tools that will come in handy later.

Professional 3D modeling packages allow you to project a texture map on objects through a number of algorithmic

JEFF LANDER | *When not circumnavigating the globe with a well-used 16th century map, Jeff can be found sitting in his comfy chair making other heretical plans at Darwin 3D. Poke him with the corner of a soft pillow at jeffl@darwin3d.com.*

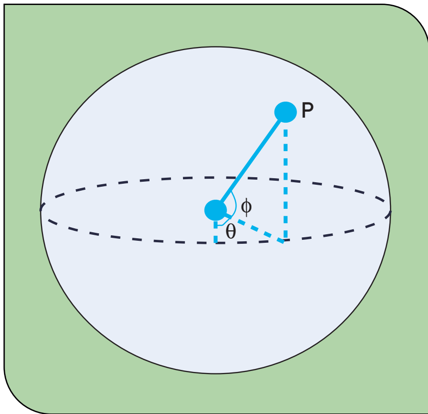


FIGURE 2. Spherical mapping.

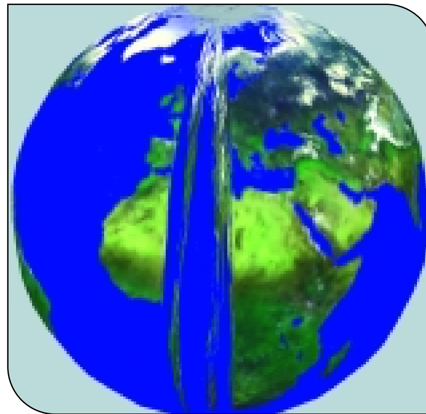


FIGURE 3. A problem at the seam.

FIGURE 4. The fixed seam.

methods. The goal of this "projection" function is to map a 3D coordinate (x,y,z) onto a 2D texture map coordinate (u,v). The function takes the form:

$$(u,v) = F(x,y,z)$$

In a 3D modeling package, you will typically see several types of mapping functions. The most common are planar, cylindrical, and spherical. Because the Mercator projection is a form of inverse spherical mapping, I'll start with that.

Spherical Texture Mapping

Given a 3D polygonal object, I want to create the texture mapping function, F. The most obvious method would be to convert each x,y,z vertex to spherical coordinates — azimuth and elevation or latitude and longitude — centered at the origin. This situation is shown in Figure 2.

I'm going to start by converting the 3D vertex, P, to the spherical coordinates (θ,ϕ). I can get the value for θ with some simple trigonometry.

$$\tan(\theta) = \frac{x}{z}$$

$$\theta = \tan^{-1}\left(\frac{x}{z}\right)$$

The value for θ will vary from 0 to 2π, but for the texture map I need it in the interval of 0 to 1. Dividing the θ value by 2π will do the job nicely.

To get the value for the ϕ, I can create a vector from the point on the model to the origin. That is easy because it is just the point itself. I then normalize that vector so it is of unit magnitude. The Y component of this vector is equal to $\sin(\phi)$. Taking the arcsin and scaling it so that the value varies from 0 to 1 gives me the V coordinate. Listing 1 contains the complete code for the spherical projection.

There are problems with the spherical projection. Like the Mercator projection, there are singularities at the North and South Poles. This is the main issue with mapping a rectangular texture onto a sphere. However, for generally round objects, such as a basketball or even a character's head, this projection works pretty well.

There is a second problem with the projection. At the horizontal seam where the left edge of the texture meets the right, I have a problem, as you can see in Figure 3.

It may be hard to see, but in the little space shown, the entire texture map is stretched across a single polygon. This happens because near the seam, the vertex calculations may generate a value for U near 1.0, say 0.9. The next vertex in the polygon may wrap to the beginning again with a value of 0.0. The hardware has no way of knowing that this polygon is on the seam and would draw the texture interval from 0.9 to 0.0 instead of the true range of 0.9 to 1.0.

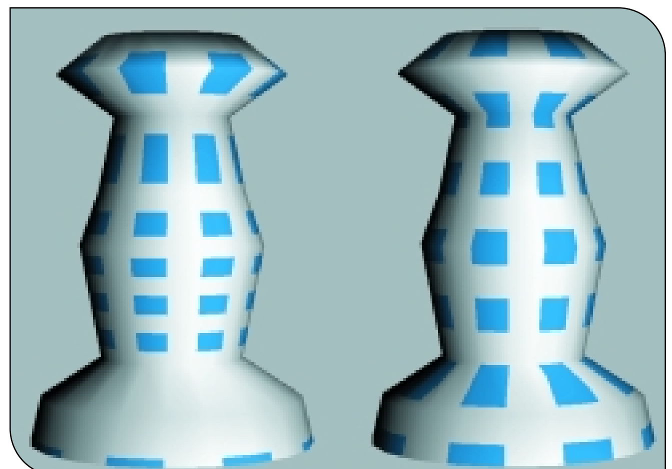


FIGURE 5. The chess piece with a sphere and cylinder map.

I don't know how other 3D packages handle this problem, but I just hack it by going back through the model, checking whether any polygon's UV interval is greater then 0.9. If so, I add 1.0 to the lower UV value. This fixes the problem, as you see in Figure 4. Obviously, if I apply the spherical mapping to an object that should have polygons spanning a large UV interval, such as a box, this hack would be bad. I make this seam hack optional, so it shouldn't be a problem.

Show Me Your Can

The spherical projection is not right for all objects. For example, if you were going to texture-map a can of Jolt cola or a chess piece, the spherical projection would not look correct. Objects like this more closely resemble a cylinder, so that is the type of mapping I want to use.

LISTING 1. Spherical projection code.

```
///////////////////////////////////////////////////////////////////
// Function:   SphereMapModel
// Purpose:    Calculate UV coordinates for model using spherical
//             projection
// Arguments:  Pointer to the model
///////////////////////////////////////////////////////////////////
GLvoid COGLView::SphereMapModel(t_Visual *visual)
{
  /// Local Variables ///////////////////////////////////////////
  int loop,loop2;
  tVector vect;
  t_faceIndex *face;
  /////////////////////////////////////////////////////////////////
  if (visual->vertex != NULL)
  {
    face = visual->index;
    // Loop through faces
    for (loop = 0; loop < visual->faceCnt; loop++,face++)
    {
      // Loop through face vertices
      for (loop2 = 0; loop2 < 3; loop2++)
      {
        // Calculate the U coordinate
        visual->texture[face->t[loop2]].u =
          atan2(visual->vertex[face->v[loop2]].x,
          visual->vertex[face->v[loop2]].z)/PI_TIMES_TWO;
        // Calculate the V coordinate
        MAKEVECTOR(vect,visual->vertex[face->v[loop2]].x,
          visual->vertex[face->v[loop2]].y,
          visual->vertex[face->v[loop2]].z);
        NormalizeVector(&vect);
        // Take the Arcsin of Y and scale to 0-1 range
        visual->texture[face->t[loop2]].v = asin(vect.y)/M_PI + 0.5;
      }
    }
  }
}
```

The U coordinate for a cylindrical map is handled in exactly the same way as the spherical map. However, the V coordinate is different. In fact, it's a bit easier, because I can set the V coordinate directly from the vertex's Y coordinate. I will need to scale the Y value and offset it a bit to get it centered. You can see a chess piece with both a spherical projection and a cylindrical projection in Figure 5. The second projection clearly has much less distortion on objects like the chess piece.

The Plane Truth

For many objects, neither the spherical nor the cylindrical projections will work. In many of these cases, though, a simple planar projection will do. In a planar projection, the texture is projected directly along one of the main world axes. This is the easiest case of all to program. To project along an axis, the UV coordinates map directly to the two remaining axes' coordinates. For example, if you want to project the texture along the Z axis, the X coordinate of each vertex becomes the U texture coordinate and the Y vertex coordinate becomes the V texture coordinate. Again, some scaling and offsets can be used to center and place the texture.

The main drawback to the planar projection is that the texture coordinates for any faces parallel to the projection axis are ill defined. While the projection appears accurate looking down the projection axis, the result looks kind of stretched on the sides, as you can see in Figure 6. The projection also goes completely through the object like an X-ray, so the texture can be seen on the back of the object as a mirror image.

To combat this problem, many commercial modeling packages allow the user to specify multiple planar projections either as multiple textures or as a single texture arranged like an unfolded cardboard box. Often this form of projection is called "box" or "cubic" mapping, where six planar projections determine each face of a cube surrounding the object. I am unsure how other modeling packages implement this feature, but my implementation is very simple. I use the face normal to determine which side of the cube will be projected on the face. This face is then used with the standard planar mapping algorithm.

Extra Features

The scaling and offset values used in the cylindrical and planar projections can be calculated automatically based on the bounding-box size of the object. This simplifies things, but I still like having the manual scale and offset controls to adjust the mapping.

In all of the projections I have discussed so far, the algorithms are aligned along one of the object's root axes. This is not a limitation of any texture projection; it just makes the example a lot simpler. I could easily change the algorithm to allow arbitrary orientation of the projection.

Editing the data is another key idea to keep in mind. These algorithmic techniques generate some UV coordinates that I can use to draw my texture-mapped objects. The coordinates may not be good ones or exactly what you had in mind, but they are a good starting point. It's certainly easier to use a base projection than to go through and set each UV coordinate manually. Many times when I use the planar projec-

FIGURE 6. Planar projection with stretched sides.

LISTING 2. Cylindrical projection code.

```
/////////////////////////////////////////////////////////////////////
// Function:  CylinderMapModel
// Purpose:   Calculate UV coordinates for model using cylindrical
//            projection
// Arguments: Pointer to the model
/////////////////////////////////////////////////////////////////////
GLvoid COGLView:: CylinderMapModel(t_Visual *visual, float scale,
                float offset)
{
  /// Local Variables ///////////////////////////////////////////////
  int loop,loop2;
  tVector vect;
  t_faceIndex *face;
  ///////////////////////////////////////////////////////////////////
  if (visual->vertex != NULL)
  {
    face = visual->index;
    // Loop through faces
    for (loop = 0; loop < visual->faceCnt; loop++,face++)
    {
      // Loop through face vertices
      for (loop2 = 0; loop2 < 3; loop2++)
      {
        // Calculate the U coordinate
        visual->texture[face->t[loop2]].u =
          atan2(visual->vertex[face->v[loop2]].x,
          visual->vertex[face->v[loop2]].z)/PI_TIMES_TWO;
        // Calculate the V coordinate
        visual->texture[face->t[loop2]].v =
          (visual->vertex[face->v[loop2]].y)*scale + 0.5f + offset;
      }
    }
  }
}
```

tion, for example, UV coordinates for adjacent polygons can overlap. That's pretty bad when I'm using the texture for a 3D paint system. When this happens and you paint on a polygon that overlaps, the paint appears in multiple places on the object — not what you really want. One solution is to allow the user to manipulate manually the UV coordinates that were generated by the algorithm. A user interface that allows you to drag UV coordinates around on the texture, as well as cut, copy, and paste those coordinates, is very handy. However, an automatic way to make sure the coordinates do not overlap would be very useful.

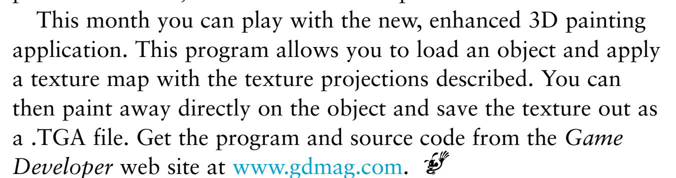
Relax, Let Us Do the Work for You

Several advanced modeling packages have begun to address some of the problems of conventional texture mapping. At Siggraph this year, I took a look at Maya 3.0 from Alias|Wavefront. In addition to all the basic texture projection options that I have described here, they have added a few new features to speed things up a bit. The first is a new texture projection mode in which the software automatically determines the optimal number of planar projection maps from a given maximum needed to texture the objects with the minimum distortion. The projected coordinates are then assembled together on a single texture map ready for painting.

A second new feature is designed to reduce distortion in the mapping. The UV coordinates generated by these algorithms do not consider the surface area of the projected polygons. For large polygons, you can often allocate a very small portion of the texture map. Conversely, very small polygons can end up with disproportionately large texture space. This is similar to the distortion of Greenland we saw earlier in the Mercator projection. To combat this problem, Maya uses a system called "relaxation" that attempts to optimize the texture-map area each polygon uses so that it is more even. I haven't had a chance to play with this much, but it looked very useful. I'm sure there are features or plug-ins for the other professional modeling packages that give you the same functionality. However, I don't know of any offhand.

Similarly, Dan Piponi and George Borshukov of Manex Entertainment showed a technique at Siggraph using Maya's dynamics simulation system to dissect and stretch a 3D object so it is completely flattened and ready for a simple planar projection. Springs at the polygon edges forced the system to maintain much of the shape and surface area of the polygons. That system should be very easy to implement using the particle dynamic methods I've described in previous columns, but I will leave that up to an industrious reader.

This month you can play with the new, enhanced 3D painting application. This program allows you to load an object and apply a texture map with the texture projections described. You can then paint away directly on the object and save the texture out as a .TGA file. Get the program and source code from the *Game Developer* web site at www.gdmag.com.



FOR MORE INFORMATION

PUBLICATIONS

Wilford, John. "Revolutions in Mapping," *National Geographic* (Vol. 193, No. 2): Feb. 1998. pp. 6–39.

Piponi, Dan, and George Borshukov. "Seamless Texture Mapping of Subdivision Surfaces by Model Pelting and Texture Blending." *Proceedings of SIGGRAPH 2000*. pp. 471–477.

Ebert, David, and others. *Texturing and Modeling: A Procedural Approach*, 2nd ed. San Diego: Academic Press, 1998. pp. 119–121.

WEB SITES

Earth Mercator projection from NASA's Goddard Space Flight Center, Scientific Visualization Studio
http://svs.gsfc.nasa.gov/imagewall/hologlobe/earth_noclouds.html

Alias|Wavefront's Maya 3.0
www.aliaswavefront.com

The Evolution of 3D Game Models Part 1

Eight years ago, just after I had graduated from college, I happened to walk into a store that sold computer games. Even though I didn't own a computer, one of the games on the shelf caught my attention. It was WOLFENSTEIN 3D: SPEAR OF DESTINY by id Software. I had never heard of it, but the graphics on the back of the box looked so cool that I ended up buying the game. I asked my brother if I could come by his work on a weekend and play it. He agreed, and I ended up playing the entire game on a Saturday afternoon. When I was done, my ears were red with excitement and I was hooked.

FIGURES 1 & 2. In just eight years, the increase of graphical sophistication in games has been stunning. Top, 1992's WOLFENSTEIN 3D; Bottom, 2000's RETURN TO CASTLE WOLFENSTEIN.

When you stop and think about how far game visuals have come in the past ten years, it is absolutely amazing. (Figures 1 and 2 show a side-by-side comparison.) Since I started working in the industry, games have evolved from 2D sprites in a flat world with 256 colors to 3D worlds beautifully rendered in millions of colors. Naturally, within this time frame all the ways in which art content is created for games has changed as well. Textures that used to be created by pushing pixels one by one are now "painted" in Photoshop with a Wacom tablet at high resolutions. The editing packages used to design the world were simple 2D editors created specifically for one game, whereas today 3D level editors such as QERadiant for QUAKE 3 are becoming more like professional modeling packages. The characters inhabiting our game worlds are also becoming more sophisticated. From the early character sprites in WOLFENSTEIN 3D and their few frames of animation, to today's characters with animations ranging from motion capture to meticulously keyframed sequences, game characters are increasing in complexity with stunning speed.

With these advances the average length of a game's development has been increasing steadily. What used to take a few months now takes more than two years. With increased production time, it's becoming harder to plan ahead and release a game with cutting-edge graphics that runs on a reasonable system specification. To hit that visual sweet spot, an art team needs to keep up with graphics trends, understand the increased sophistication with which models are built, and make the necessary adjustments to ensure their game will look good two or more years in the future. To accomplish this, an art team not only has to understand the current methods used to create models, but also be visionary in planning for the next generation of games.

FIGURE 3. Lara Croft and her early contemporaries redefined how characters looked and interacted with 3D environments.

The First Characters With a Backbone

With the advent of the Playstation and Nintendo 64 consoles and 3D graphics cards for PCs, it became clear that 3D was the way of the future. Games became more immersive and characters came alive with increased expression and personality. Games such as TOMB RAIDER and MARIO 64 introduced characters that could interact with their environment like never before. Their characters had smooth animations that oozed personality, as Lara Croft demonstrates in Figure 3, helping to establish them as true icons. Most importantly, the introduction of 3D game characters created a new level of immersion for players.

The first 3D character skeletons all

MAARTEN KRAAIJVANGER | *Maarten is the lead artist at Nihilistic Software. Prior to working on VAMPIRE: THE MASQUERADE — REDEMPTION, he was at Cyclone Studios where he worked on CAPTAIN QUAZAR, BATTLESPORT, and REQUIEM. He's currently trying to set the target on the visual sweet spot for Nihilistic's next game. Contact him at maarten@nihilistic.com.*

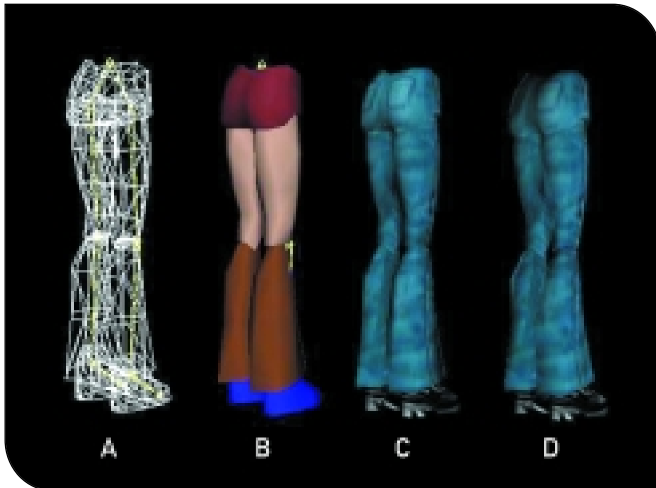


FIGURE 4. Adding lighting reveals the segments where different areas of the model's geometry meet.

shared some of the same attributes. The characters were divided up into separate parts to match the different joints in the body; both Lara Croft and Mario were built in this manner. In Figure 4, the model shown is composed of various pieces of geometry: rear end, thighs, lower legs, and feet (A shows the model in wireframe, and B shows the flat-shaded model). Now look at the same geometry with a texture map applied (C). Because it has flat lighting, the model comes out looking good. Much like when you play TOMB RAIDER in software without a 3D accelerator, the model in C looks solid and has no apparent flaws. However, when dynamic lighting is added (D), the limitations of the model become more apparent. Because of the segmented method in which the model is created, the lighting on the different parts highlight the areas where the different pieces of geometry meet. Although this can be reduced with careful model design and construction, it is almost impossible to create realistic skin in places where joints come together, such as knees or elbows. The lighting will invariably draw attention to imperfections in the intersection of components.

Another limitation of the early 3D models is that they required very careful construction at the joints. Because these parts needed to animate, the joints' geometry had to be constructed to prevent any part from poking through. Additionally, each piece of geometry needed a cap so that you couldn't see into the parts. This could be a time-consuming process, because all the joints had to be rotated to their extremes to ensure that the geometry stayed together.

Once the model was carefully built and attached to the skeleton, it still had one main limitation: because the characters were made up of many parts, when you applied detailed textures, the textures broke up when the model started animating. The Lara Croft model avoided the texture problem by using mostly solid colors. It worked well for the game, because Lara was never that big on screen, and looked great from the player's point of view. The limitations of the models became more apparent during in-game cinematics, in which the camera is so close to the model that the player notices both the lack of textures and the building blocks that make up the model. A more advanced technique was needed to make the models look realistic.

Characters That Bend

The next step in the evolution of character models made it possible to do a better simulation of skin, even when the camera got close. For skin and cloth simulation to work, the model had to stay whole rather than be broken up into many parts. By staying whole, the skin and cloth looked and behaved more realistically. HALF-LIFE did a great job with this. It had scientists who wore lab coats and characters that could talk with an actual moving mouth. HALF-LIFE introduced a new layer of complexity to character models that had some great benefits. However, there were still limitations.

To keep a character whole and make it work with a skeleton, the character needed to be set up with clusters; in earlier methods of setting up 3D characters, the individual polygon meshes were attached to the joints. The next step in the evolution of a model required a different type of attachment of the polygon mesh to the joints. A polygon mesh is made up of vertices, which can be grouped into clusters attached to the joints; a cluster is a subset of vertices that makes up the mesh shape. These groups of vertices then move according to the joint rotation, but the entire mesh shape stays whole. With the introduction of clusters, a better simulation of skin became possible.

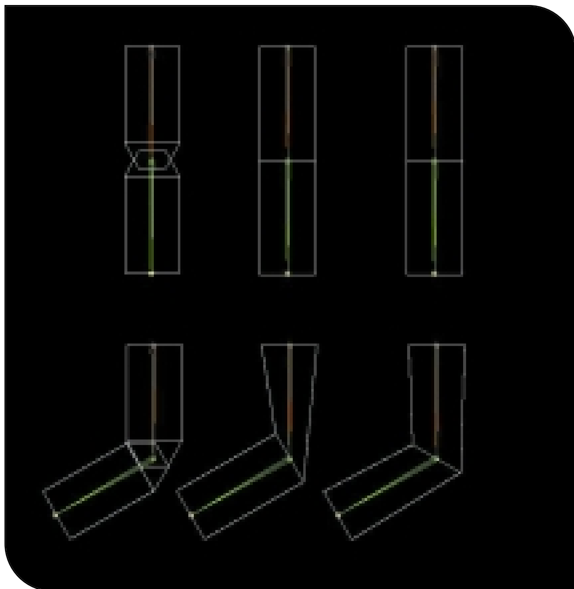
The benefit of binding a model to a skeleton using clusters is that it gave the artist the freedom to build the model as it should look. Once the model was complete, the entire model mesh could be attached to a skeleton, and you didn't have to worry about breaking it into separate pieces. In HALF-LIFE, the models looked whole since they were made up of one main mesh and no longer had lots of intersecting parts. Another benefit of a continuous mesh was that that the model could be textured more realistically, and the textures would not break up when animated. The models for HALF-LIFE were able to look convincing even when you walked right up to them and looked at them up close.

An additional benefit of cluster-binding a character model was the ability to open and close the mouth using the head mesh.

FIGURE 5. HALF-LIFE's models sported more realistic skin, with cluster-bound continuous meshes on the skeletons.

HALF-LIFE was one of the first games to use convincing real-time lip synch for their characters. This additional layer of realism gave the characters in HALF-LIFE a more believable feel and in turn helped make the player feel more immersed in the game. Although the mouths worked well, the restriction of the clusters was apparent. When the jawbone opened, all the vertices attached to the jaw opened the same amount. The corners of the mouth did not stretch; instead, the mouth moved a little bit robotically.

Even with the benefits of the introduction of clusters, the limitations were visible. When the joint of a HALF-LIFE skeleton rotated, the vertices rotated the exact same amount as the joint's rotation. In the real world, skin and cloth move differently, depending on such properties as elasticity, thickness, and consistency. Because the artist did not have control over the amount of movement of the vertices near the joints, they tended to pinch when rotated. So when an arm or knee bent, the inside of the joint actually became much thinner. If the artist set up the model's geometry carefully around the joints, the pinching could be reduced. To fix this, the artist needed to be able to have more control over the vertices when they bent. The model had to be set up with weighted vertices.



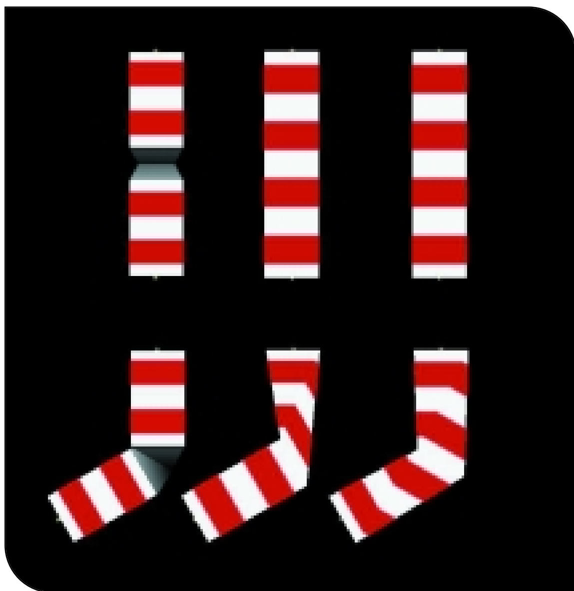


FIGURE 6 (top). Three different joint attachments in wireframe. The joint on the left is two separate pieces of geometry, in the center is a joint built with clusters of vertices, and the joint on the right was built with weighted vertices.
FIGURE 7 (bottom). The same joint setups with textures applied.

Throwing Some Weight Around

Once characters' polygon meshes were subgrouped into clusters, they looked more solid, since they were made of one piece of geometry. But to reduce the pinching problems and take the next step to realism, models needed weighted vertices. In VAMPIRE: THE MASQUERADE — REDEMPTION, the game I worked on over the past two years, we used cutscenes extensively with lots of close-ups of the in-game models. In order for the models to hold up under such intense scrutiny, we implemented weighted vertices. In addition to making the models look better, it ended up saving us a lot of time because weighting eliminated a lot of time-consuming tweaking. Weighting the vertices smoothed areas where pinching occurred and let us continue with our work. After we got comfortable with it, using weighted vertices also helped us achieve more realistic facial expressions.

In Figures 6 and 7, you can see the three different attachments and how they affect geometry, both in wireframe and with a texture applied. Lara Croft used the attachment method shown on the left. The separate pieces of geometry are attached to the joint and move separately when the joint bends. The middle example uses simple clusters without weights, like the setup used in HALF-LIFE. The geometry looks whole, but when the joint rotates, the vertices at the joint start to pinch and lose their shape. The example on the right shows a model with weighted vertices. The weighted vertices only move 50 percent of the total rotation of the joint. With this type of control, the skin and cloth of an object become more manageable.

Another great benefit of using weighted vertices was that it added more facial control to the models. Weighting the vertices allowed us to simulate the muscles underneath the skin. In VAMPIRE, we created expressive faces using joint attachments with the weighted vertices. In our main character, Christof (Figure 8), you can see the results of weighting the different parts of the face. The vertices along the eyebrows have different weights that simulate the contraction of the forehead muscles. Rather than having the eyebrows move uniformly, the eyebrow can be raised and twisted, yet look realistic. To create a sneer, a joint placed on the top lip can have a cluster group of vertices that go halfway up the nose. When the joint moves, the farther the vertices are from the top lip, the less they move. So, the vertices on the lip move 100 percent of the joint's translation, the nostril moves about 50 percent of the translation, and the ridge of the nose moves just slightly, with only 10 to 20 percent of the total joint rotation. With these weights, we were able to simulate the muscle contracting the lip, creating an effective expression of disgust. By attaching parts of the face, such as the corners of the mouth, lower lip, eyelids, and eyeballs to

FIGURE 8. Weighting different parts of Christof's face allowed for a variety of more natural-looking expressions.

joints and carefully weighting them, new types of expression became possible.

Each of the attachment systems has limitations, however. One thing HALF-LIFE didn't have to deal with was extensive shoulder animations. The shoulders are one of the most extreme joints in the body, so when one is creating a game model, the shoulder can become an issue if it needs a wide range of motion. Because the shoulder is a ball-and-socket joint and extremely flexible, attaching a solid mesh to a skeleton can be really hard, even with weight control over the individual vertices. (Baseball game developers out there must know what I'm talking about.) In VAMPIRE, the characters needed to swing lots of different Dark Age weapons, so we needed a full range of shoulder motion. Although we had good results and the shoulder was able to keep its shape fairly well when it moved, it was still far from perfect. The shoulders turned out to be a big time-sink since they had to be carefully weighted and tweaked to look good no matter what their position. This became even more obvious when the character starts off shirtless, and the player can see how the shoulder moves. With one of our very muscular characters, the dense geometry that we needed in the shoulder to give it the right shape prevented the weights from working correctly, and we still ended up with awkward-looking shoulders. In retrospect, we realized the problem: the skeleton did not simulate a real shoulder. We only created a ball-and-socket joint, rather than try to simulate the entire shoulder blade and clavicle.

Even with their great benefits, weighted vertices still have limitations, and new attachment systems have to be explored. As with all these different methods, the artist actually working on the model and the attachment is going to have a bigger impact than the method itself. But pushing technological barriers and giving artists more freedom to create better models can lead to better results.

The Characters of the Next Generation

With an understanding of the three evolutionary phases of character setup, artists are able to make art for today's games. The problem is, we're currently making games for two years in the future. If you don't evolve with the times and keep pushing for even more sophisticated ways of creating models, you'll find yourself going the way of the dodo. The next-generation consoles are poised to come into more people's homes than their predecessors. Ultimately, with increased budgets and longer development times, there will be fewer games released, and making your game stand out from the competition is going to be even more difficult. The time has come for us to push the envelope and explore model-creation techniques that have until recently only been used in film. So far, game art has been following the road paved by the artists in the movie industry; the next generation of models could very well be competing with them. In next month's article, I'll focus on some of the new features that will be introduced in the next generation of character models.

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

The Art of Noise

Game Studio Recording and Foley

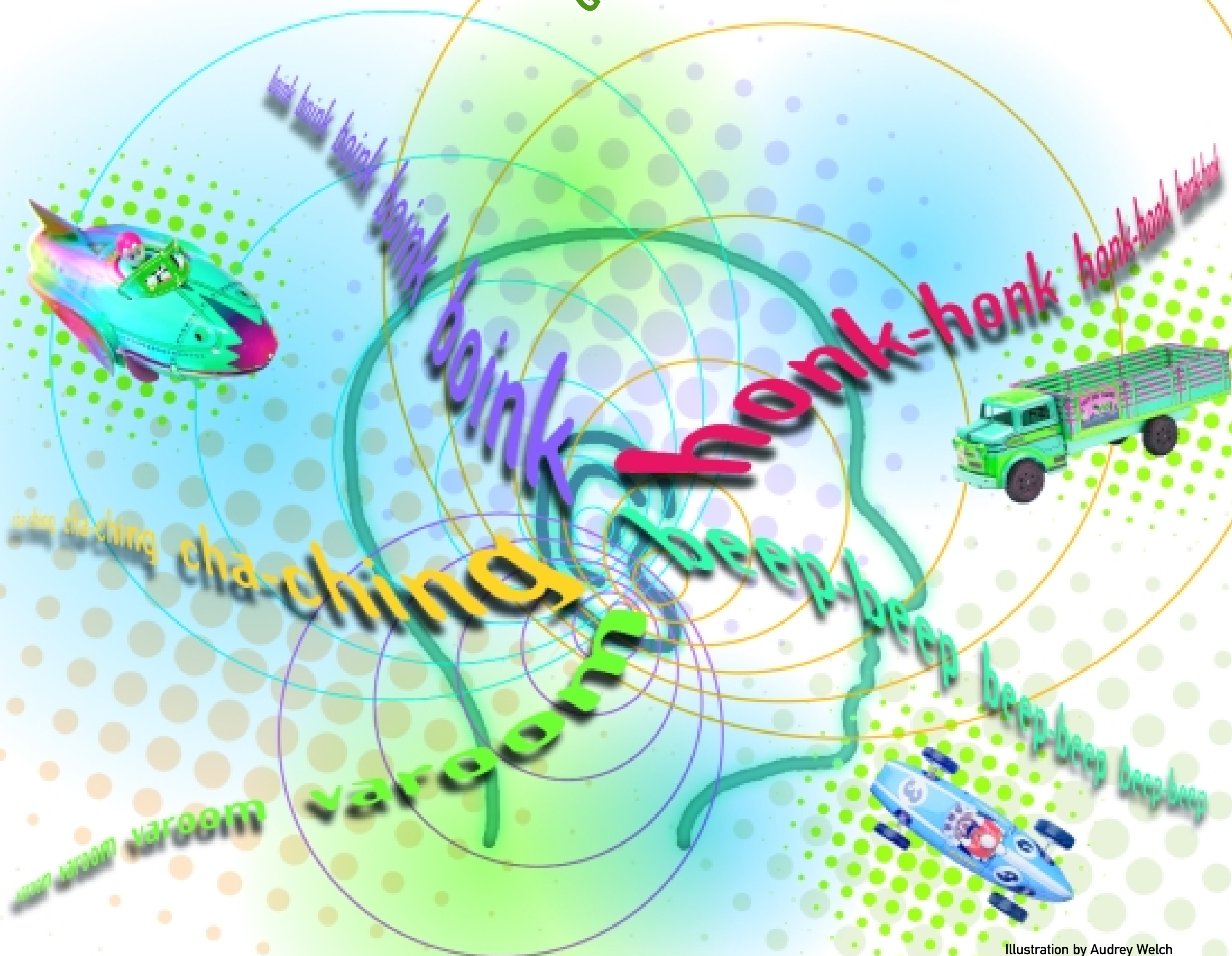
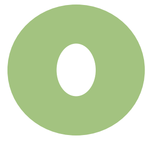


Illustration by Audrey Welch



On the increasingly level playing fields of 3D acceleration, elaborate input devices, and broadband capability, game companies are working harder than ever to find ways to differentiate themselves from one another. While design and art play a major role in this process, sound is also an important factor in a game's success or failure. In fact, it's increasingly rare to find top-selling titles that don't also feature top-notch audio design. So if you're having trouble making your game's sound environment bigger and more distinguishable from your competition's, this article is designed to help.

Sound designers have myriad tools and techniques at their disposal to create a complete audio environment for their game. Sound effects libraries are often the easiest starting point for a design job and are available in dozens of flavors, from big production houses down to small, one-man shops. In general, stock sounds are available on individual CDs or in complete sets with a common theme. Some of the more progressive sound providers offer download capability from their web sites, allowing you to grab samples even while pulling an all-nighter.

Sound generation from an electronic source, such as a synthesizer or a software package, is another route to take. Sound designers working off of traditional white or pink noise functions, which provide an even noise distribution over a frequency range or octave values, have often found it the most flexible way to sculpt a complex sound.

Field recording is somewhat similar to studio recording, except that in many cases you will not be in direct control of the source material, but will rather be working around it. Field recording can be especially useful for recording large scenes and ambient waveforms. Issues to consider when field recording are often obscure. For example, at a military base, the recording of some equipment may not be allowed without permission. Even worse, some environments may have electrical equipment that will interfere with your gear's recording capability.

While not a replacement for other sound design techniques, in-studio recording of sound effects (called Foley) saves both time and money spent searching for the right material elsewhere. For cutscenes or streamed audio clips, Foley can be entirely comprehensive and extremely fast. Additionally, material can be created that is perfectly matched to your production, not shoehorned in from an audio CD. Even today, it is not uncommon to hear some of the same tired sound effects reused in game after game and in commercial after commercial. The determined walk-step of your gun-toting hero can be cast a certain way, while the sneaky footsteps of the villain he is about to fight can be cast in another. Finally, Foley techniques provide an excellent excuse to junk up the development studio with all kinds of fun, eclectic props (see sidebar, "Foley Props").

This article is intended for sound designers who are looking to go beyond the "sound via CD" techniques commonly overused in game production. The focus is on studio recording within typical game development budgets. Experienced sound designers already working in upscale studios may also find this article a good refresher.

ROBERT STEVENSON | *Robert is the vice president of production at iROCK Interactive, a music game development company in North Carolina. When he's not making games, he's busy playing them. He can be reached for questions or comments at robert.stevenson@irock.com.*

Setup

Although in-studio recording is a fairly involved process, setting up correctly before a session can be timesaving. In fact, a lot of things ought to be done before the first audio take occurs.

There are no hard and fast rules for budgeting sound design, other than to make sure you do actually budget for it. Too often on small projects it's listed as something the sound programmer or the music guy can do on the side, or in some cases it's forgotten altogether. It is also good to tackle sound development early enough in the project to make sure it nears completion long before the headaches of trying to master happen. Using an outside contractor for your audio development is recommended if you can't do the work appropriately in-house. Bringing them on the project early will also yield fewer hassles and better results, as there will be more time to make everything perfect.

The specific time requirements for sound effects development will vary from designer to designer and project to project. Variables such as how complex the sound requirements are, the sophistication of the audio equipment being used, and how much pre-existing library material is already on hand also enter the equation. For a typical game development situation, a good rule of thumb is to budget about half an hour per incidental effect (minor or background-type material) and a full hour for major elements (foreground and repeated sounds). Foley work can expedite this process, particularly for minor effects.

Budgeting equipment is another issue to consider. Starting from scratch, you can expect to put together a simple game-capable recording studio for both vocals and effects for under $5,000. Adding high-end, feature-film-quality equipment or a multichannel digital recording system can run you quite a bit more. If your sound budget is tight, consider renting sound equipment rather than purchasing it. Items such as microphones, mixers, and DATs typically rent for nominal rates and will often be cheaper on weekends.

For planning an in-house recording session, a good starting point is to create a comprehensive sound list or an audio storyboard for the game. From an accurate sound list, it isn't difficult to break sound groups into rough production categories of how each waveform will be built. Foleys are generally best for "personal" sounds attached to characters and creatures, or specialty effects that require some individuality to set them apart in the game. Cutscenes or animations are also prime candidates for custom recording work.

Outside of the sound list, you will need some type of recording report to track what actually happens during the recording session (see Figure 1). While session

notes can be scribbled on a piece of scrap paper, the time it takes to create an official comprehensive reporting system will be more than made up when you are ready to edit and mix down your effects. A typical report will have the sound or sound group name, along with take numbers, the presence of the right and left audio channels, and some basic equipment setup information. The recording report should also include some type of indexing scheme that matches your equipment or saving process. Also be sure to leave room for miscellaneous notes on each take. With a well-planned recording report and a comprehensive sound list available, all you need is something to record.

Through the years, sound designers have relied on all manner of noisemaking objects (dubbed props) and tricks to create the sound they are looking for. When the actual object needed for recording isn't available or is too unwieldy for the studio, creative improvisation often works best.

It's a good idea to start a prop box filled with all kinds of noisemakers long before a recording session takes place, preferably at the start of a project. Props can be almost anything from baking soda to stalks of wheat; the only limit is your imagination. In situations where a prop might be destroyed during the take, such as the shattering of a glass, make sure to get multiples for practice or mistakes. It's also a good idea to have on hand various wearable prop items in your collection of goodies. The sound of a trench-coated hit man drawing a pistol from his pocket is going to be a lot more convincing if a trench coat is actually worn in the take. The same notion applies for shoes, armor, helmets, jewelry, and specific types of fabric, such as silk and wool.

Foley pits are another addition to your recording area if you anticipate needing to do footsteps. Essentially these pits are lowered sections of floor filled with different walking surfaces. Sand, rock, water, and tile are all common pit fills. A simple inexpensive pit can be constructed out of a deep two-by-four frame (see Figure 2). Materials you might work with, such as concrete, flagstones, metal grating, and even sod, can be purchased at almost any major hardware store.

The space in which you record a sound is almost as important as the sound itself. If you don't have access to a sound recording facility or are working within a tight budget, almost any moderately sized room, preferably one that is isolated, can be adapted to the task. The most pressing concern with a recording room is controlling the acoustics and, if possible, using them to your advantage. Problems generally arise from sound bounces, where a sound wave traveling from its source is reflected off a hard surface, such as a table or wall. In a small space, these reflections are usually not perceptible to the human ear; in larger spaces they can manifest themselves as an echo or multiple dispersed echoes, called reverberation.

These alterations to the sound, sometimes referred to as coloration, can often be dramatic, not only betraying the characteristics of the recording space, but also modifying the fundamental characteristics of the sound itself. Luckily, a little knowledge of sound waves and the physics involved can help eliminate such problems before they start.

Sound waves are normally created through the elastic compression (the rapid

FOLEY PROPS

BBs. Classic BBs or peppercorns come in handy for everything from rainfall to shotgun pellets.

Baking sheets. Great for banging sounds like the metal sides of a vehicle or for creating metallic thunder effects. Aluminum foil and sheet metal can also be used for variations.

Cat litter. Many of the dry, rounded types can simulate rocks sliding or scraping. Walking on it produces a sandlike effect.

Lettuce. Most vegetables are great for classic crunching and flesh-ripping sounds. Try snapping a piece of dampened celery for a nice bone break.

Cardboard tubes. Excellent for swishing sounds. Purchased with an end cap, they can simulate mortar shots or cannon fire.

Compressed air. Air canisters (even balloons) of all types can create explosions, pipes snapping, or rocket exhaust. Releasing compressed air in a water-filled trashcan produces some interesting effects.

Fabric. Expanses of various kinds of fabric can make anything from flags flapping to parachutes opening. Try combining stiffing elements to create dragon wings.

Glass. Scraping glass on glass yields all kinds of interesting effects, from creepy horror sounds to mechanical grinds.

Kitchen devices. Small electric appliances (can openers, blenders) coupled with dampening materials can make terrific engine, platform, or sci-fi sounds. Electric razors can also work wonders.

Military gear. A visit to the local surplus store can turn up all kinds of interesting hand props and game-ready devices, from ammunition cases to wearable harnesses with clips and pouches galore.

Paper towels. Wet paper towels are great for stepping on or slinging mud.

Phone books. Almost any kind of dense book is perfect for hard falls or solid punching impacts.

Sound Recording Report		Game:	Date:	Settings/Configuration:
Sound Group	Index No.	L Channel	R Channel	Notes:

FIGURE 1. A recording report tracks what happens in the studio, and is an excellent reference later on during editing and mixing.

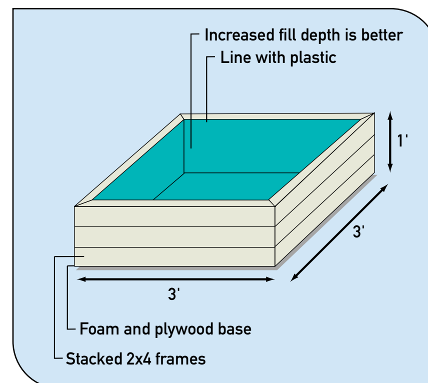


FIGURE 2. An example of a "budget" Foley pit setup, ideal for recording footsteps on a variety of surfaces.

pushing together) and rarefaction (the rapid pulling apart) of air molecules (see Figure 3). While one phase of compression and rarefaction is called a cycle, the physical distance each cycle covers (noting that sound travels at roughly 1,130 feet per second) is referred to as its wavelength. The number of cycles that occur in a single second are measured as a frequency in hertz (Hz). Humans with exemplary hearing can detect frequencies from about 20Hz to 20kHz, although you'd be hard pressed to find any modern audio standard delivering the entire range.

Because sounds are physical waveforms moving through air, they are free to intermix. When two sound waves encounter one another, their interference can be termed either "in phase," where both waveforms are perfectly aligned, or "out of phase," where the two waveforms are shifted (phase-lagged) apart. The result is a new waveform based on the sum of the displacement of each contributing waveform (see Figure 4). In the two most extreme cases, perfectly in phase and exactly 180 degrees out of phase, a pair of sound waves will double in amplitude or completely nullify each other, resulting in no sound. Either case is a recording mess.

Sound waves have other important properties to consider when setting up a recording session. Each waveform's compressions and rarefactions also have a strength known as the intensity. The spread of values from minimum to maximum intensity is the sound's dynamic range. When a sound wave radiates from its source, it starts with a uniform intensity pushing on a volume of air. As it continues to travel away from its source, the expansion continues, but it has to push increasingly large volumes of air to achieve the same result. Without amplification, the sound's acoustic pressure dissipates, decreasing its intensity. The same behavior can be observed in any waveform outside of a vacuum. For example, as the shock waves from an earthquake spread from its epicenter, the area affected increases while the relative strength of the shock waves declines.

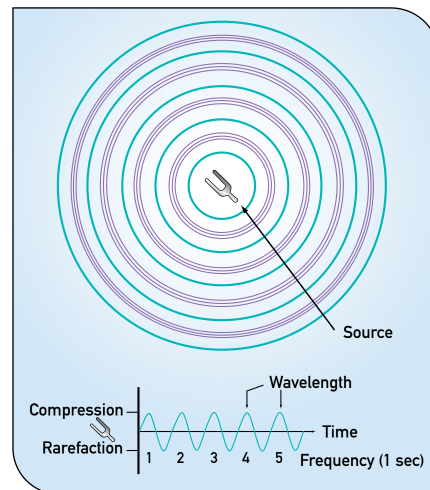
Sound pressure varies inversely with the square of the distance from its source. In numeric terms, an audio wave's intensity drops by six decibels (a logarithmic measurement of a sound's loudness) with each doubling of its radial distance from the source. This exponential relationship is known as the inverse square law, and keeping it in mind during any recording process can be extremely helpful. By increasing the distance from your microphone to the closest source of reflections (at least two to three times greater), you can dramatically diminish the intensity of any audio coloration (see Figure 5).

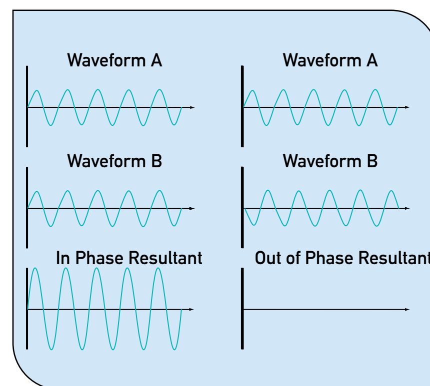
Of course, providing increased distance from the source of reflections is not the only thing you can do to ensure a clean (sometimes called a "dry") recording. Soundproofing the recording space from outside interference is a good start. While full sound studio setups can run into the millions of dollars, small changes to any space used for recording can make a big difference. Unless they were designed for acoustics, rooms with windows are not advisable. Any doors in the recording space should be replaced with the solid-core variety and the walls should be soundproofed with insulation. Felt-padding or weather-stripping door frames and air vents can also help eliminate vibrations caused by air pressure variations in a building.

Music supply companies sell wide varieties of diffusers, used for dispersing an audio reflection evenly, and dampening tiles, used for trapping sound. Hanging these on the walls of your recording space near the closest reflection areas to your audio source or in the direction of the microphone's pickup (see below) can work dramatically well. If you're on a very tight budget, consider purchasing acoustic blankets. More often found at theatrical production houses than music stores, they can be hung or draped about for adequate effect.

Microphones

Perhaps the most important piece of hardware in the recording room is the microphone. While there are literally hundreds of manufacturers, types, and styles, they can be broken down into a few basic categories. A good Foley setup will typically have a wide range of microphones available. Sometimes it may even be useful to presort microphones into smaller groups





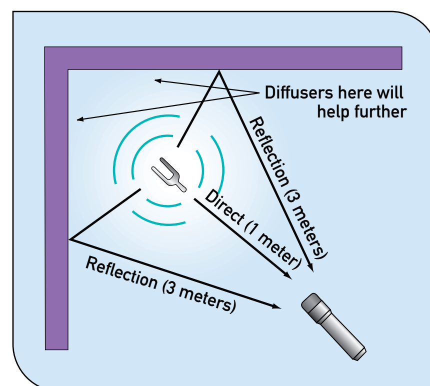


FIGURE 3 (top). How sound waves work. A little knowledge of the mechanics involved will help make your recording sessions successful.
FIGURE 4 (center). Different waveforms can combine with varying results. On the left, two different waveforms combine to produce an "in phase" resultant, on the right, the resultant is "out of phase."
FIGURE 5 (bottom). The inverse square law can help you diminish audio coloration through proper microphone placement.

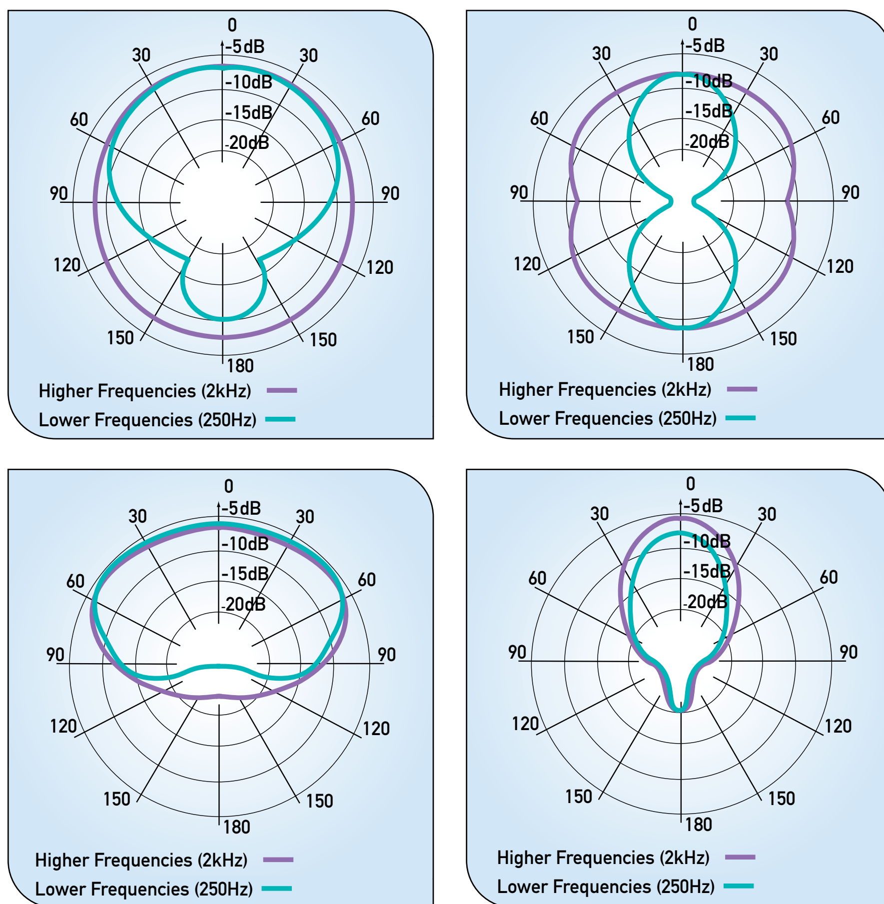


FIGURE 6 (top left). An omnidirectional microphone has a spherical field pattern, ideal for when background noise is meant to be part of the recording. FIGURE 7 (top right). A bidirectional microphone picks up sound equally from both the front and back of the mike. FIGURE 8 (bottom left). Cardioid, or unidirectional, microphones are used commonly in Foley work. FIGURE 9 (bottom right). Hypercardioids offer increased directionality over regular cardioids, but can be tricky to work with.

that complement one another.

The process of taking compressed waveforms of air molecules and turning them into electrical signals is the function of a microphone's pickup. Pickup mechanisms are often located near the tip of the microphone and are usually encased behind some type of acoustically transparent protection.

Dynamic pickups are older, simple electromagnetic devices that operate off the vibration of a polyester diaphragm. As pressure waves strike the diaphragm, an attached conducting coil moves back and forth axially along a stationary magnetic field. The movement of the coil through the field induces an electric current directly proportional to the amount of vibration in the diaphragm.

Dynamic microphones make great general-purpose devices, good for recording low-frequency or especially loud material. Good dynamic mikes will typically have a frequency response range from 50Hz to 17kHz. Musicians often use them to record drums.

Condenser microphones, sometimes called capacitor or electret mikes, are a more recent invention and operate differently from their dynamic cousins. The diaphragm for a condenser microphone is some type of charged metallic surface mounted adjacent to a fixed, oppositely charged plate. As sound waves vibrate the diaphragm, the gap between the plates alters, allowing some electrons to jump from one plate to the other, which varies the capacitance and charge, yielding a small electric current. This current is so weak that it must be amplified before it exits the microphone. Therefore, condensers have a small preamplifier mounted in the mike body just behind the pickup. Because of the diaphragm mass and sensitivity differences, condensers absorb a broader frequency spectrum than dynamic mikes, ranging from as low as 20Hz to as high as 24kHz.

With traditional condensers, there is a need for a separate (sometimes called "phantom") power supply to actively charge the diaphragm plates, which makes them a little cumbersome for field work. Electret condensers, a special subtype manufactured with a chemically polarized diaphragm, do not require additional power and so avoid this problem. Condensers of all types are not the most rugged microphones, and care should be taken when handling them. Nonetheless, condensers make excellent Foley mikes, allowing the widest range of sound to be captured, giving your work an accurate and rich feel.

In addition to the pickup, microphones also have a sensitivity pattern that governs where they can gather aural input from. As their name suggests, omnidirectional microphones have a spherical field pattern radiating out from the diaphragm (see Figure 6). This makes omni mikes useful when background noise or acoustics are intended to be in the mix.

A bidirectional microphone picks up source material from both the front and back equally well (see Figure 7). Sound coming from the sides will be only minimally captured, if at all. This configuration is sometimes dubbed "figure eight" because of the shape of its field pattern.

Unidirectional, or cardioid, microphones are sensitive front-diaphragm devices usually favoring higher frequencies in their directivity. They are useful in stage and voice work, where they can be aimed at the sound source, and pointed away from the audience (see Figure 8). In Foley work, cardioid microphones are very common because they offer directionality with the freedom to move around.

Hypercardioid microphones feature longer mounting tubes and increased directionality over traditional cardioids. Taken to the extreme, these microphones are

called supercardioid or shotgun mikes. These rifle-like microphones have a pickup pattern that matches their name — essentially straight out in front (see Figure 9). Because of their high sensitivity, they are excellent at picking up sound within a very narrow field while eliminating nearly everything to the sides and rear. This can be incredibly useful in situations where the sound source is statically located in front of the mike, such as using guns or hand prop manipulations. However, in a situation with a moving source, such as an actor or a large Foley prop, shotgun mikes become unwieldy and difficult to track with, even for veteran operators. It's also worth noting that no matter what you may have seen on TV, typical shotgun microphones don't have any more range than their cardioid brethren.

From booms to woolies, there are all kinds of microphone accessories available on the market. While most of these accessories are not particularly useful for a typical Foley shoot, a good set of stands and a shock mount are often handy to eliminate vibrations or the occasional microphone trauma. It is worthy to note that experimentation with microphone coverings, such as foam, socks, or even condoms, can sometimes produce interesting dampening effects, if desired.

Microphone pricing varies wildly, ranging from low-end models around $50 to high-end ones up into the thousands of dollars. While you generally get what you pay for, sometimes the best sound for a particular session can be had with an inexpensive, off-the-shelf mike. It's best to experiment and find a set of microphones that work well with your particular recording environment. Every recording task has a specific set of requirements, and by having a wide selection of microphones from which to choose, you can select one that is best for the job.

Recording

Capturing a sound with microphones is an art within itself. The difference between a "big" sound and something lesser is often due to the placement of the mikes during a take. While dedicated practice will ultimately prove to be the best teacher, understanding some basic principles and common techniques before pressing the record button can help you get off to a good start.

In most cases, for game audio you will want to record monophonically. However, depending on the output specification, you may want a stereo recording. While stereo microphones do exist, it's relatively easy and more flexible to achieve excellent, coloration-free results from two monaural microphones. Knowledge of a few basic arrangement patterns and a little experimentation is enough to get started.

The easiest way to make a stereo capture is to arrange the microphones equidistant from the signal source. This can produce a decent, wide stereo effect on large sources or be useful for capturing multiple points of view, but can cause problems if wave cancellation issues arise, particularly in a dynamic shot. A particular disadvantage to microphones spaced apart is that they will not accurately capture the signal the way human ears naturally hear it because the interaural difference (the difference in physical spacing between channels in a stereo field) is too large. For that, you need to place the microphones in a pickup pattern that makes them sample both the right and left versions of a single, coincident point in space.

Traditional coincident patterns will vary from setup to setup, depending on the microphones used. Typically, two cardioid or (if more ambience is desired) bidirectional mikes will be oriented perpendicular to each other, split at 45 degrees and facing the sound source.

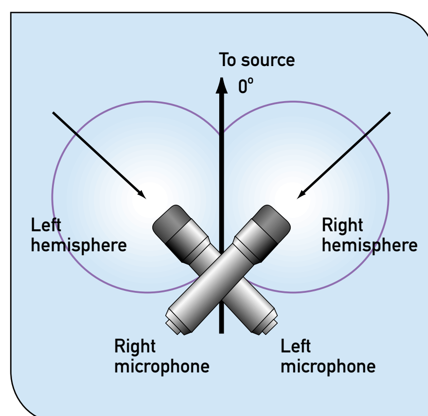


FIGURE 10. An XY pickup pattern mimics the behavior of a single stereo microphone by crisscrossing two monophonic mikes approximately 90 degrees near the tips.

The XY pickup pattern is a common example of the coincident technique using two identical monophonic microphones. The idea is to mimic the behavior of a single stereo microphone by crisscrossing the two independent mikes slightly more than 90 degrees near the tip (see Figure 10), trying to keep them close together. Keeping the mikes arranged in this fashion also helps minimize any phasing artifacts caused by the sound reaching the diaphragm of one microphone before the other.

The ORTF pickup pattern is an extended modification to the XY technique. Simply stated, this pattern is a precise 110-degree separation angle between two microphones spaced 7 inches apart, similar to a human head. The MS (mid-side) pattern is another common arrangement that uses a forward-facing cardioid mike to

RECORDING NOTES

• Avoid electronically manipulating the sound during recording. Adding compression or reverb later will be just as effective, and if something goes wrong you'll always have the original source to fall back on.

• Avoid recording on an empty stomach. Your stomach will invariably growl during the best take of a set.

• Watch out for unintended clothing noise. Remove anything that could cause unwanted ambience, such as watches or jewelry.

• Always monitor the signal. Unfortunately, ears exposed to the sound source aren't a good indication of what actually got recorded. Listening through a pair of high-quality headphones is the ideal monitoring technique.

• Don't forget to report every take. This should become natural. Not doing so can cause serious headaches after the session.

• Record to a medium outside of your editing environment if possible (sampler, DAT, minidisc, and so on). You'll always have a backup copy when you're working.

• Be attuned to background noise in your recording environment. Computer fans, air conditioning, and even co-workers down the hall can leak in, ruining a recording.

capture the source directly, and a perpendicularly facing bidirectional microphone to capture any ambiences and reflections. Playing with the input level on the microphones in the MS pattern will produce a narrower or wider stereo field.

In addition to microphone orientation, you will need to select an appropriate distance from the source to record. This is sometimes called the recording perspective. Close-in approaches generally yield a tight focus with little ambience, while distant techniques will pick up the surrounding acoustics. In cases where you are trying to capture a detailed effect, such as a door latch or machine gun shells hitting pavement, it's generally preferable to work close in, keeping the mike gain at a minimum. Equalizing in postproduction on a tight recording can make the sound dramatically bigger. Choosing a more distant positioning may be required, depending on the type of microphone you are using and the size of the sound source.

Although you may have already eliminated major sources of reflections within your sound space, sound bounces off small surfaces close to the microphone can often change the tonal quality of the recording. Common offenders are nearby tables or improperly positioned stands that may be in the work area. Luckily, testing for these close reflections is easy, and the fixes are even easier.

By placing a small mirror on the suspect surface and eyeballing it from the microphone's point of view, you can quickly tell if the sound is bouncing off the surface from the source. If the sound source is visible in the mirror, the object is likely causing the unwanted reflections. Reorienting the offending fixture to make sure the prop is not visible in the mirror is the easiest solution.

Now that you've got your props and microphones set up, you'll be tempted to start recording like mad. But before you start rolling, make sure to do some practice runs to avoid obvious problems such as clip-outs, where a sound's dynamic range has exceeded the capabilities of the recording hardware and is experiencing a distorted cutoff. If possible, also record a calibrated half-minute lineup tone to help you, or someone else that needs to work with the session material later, to be able to properly configure the equipment. If enough planning has transpired, working through even a long sound list should be relatively quick.

Adding Zchwing!

Once you've got days of recordings captured and each effect done six different ways, you're bound to find you're still missing a little something. The process of intermixing, resculpting, and adding highlights to sound effects is often called "sweetening," and doing so can often turn a so-so effect into one that really roars.

As a rule of thumb, it's best to start by having your multi-part sound clips available in a multi-track editing tool, or in the case of a two-channel editing tool, at least in a mix-ready state before starting any enhancement. For example, a shotgun effect being built for a first-person shooter might start with a cocking sound followed by a trigger effect, and then multiple simultaneous blast effects spanning the entire range of the audio spectrum, each within their own distinct tracks. With the correct setup, all parts of the sound will be on the same acoustical page and in time with each other, making further work easier.

When working with sound effects, it's important to understand the tonality of other audibles they are going to appear with, particularly during cutscenes where there should be a great amount of control over the audio presentation. Avoiding range conflicts caused by having too much acoustic information within a limited part of the frequency spectrum is a crucial task for good sound design. Fortunately, equalization can provide more punch to your effects to push them past the music or vocals as appropriate.

All equalizers are variations on the same theme, producing better-sounding audio by increasing and decreasing the intensity of frequency bands within the audio spectrum. Common types of equalizers are graphic, which allow partial or full octave control over bands within the frequency spectrum, and parametric, which allow for

SOUND ADVICE FROM THE PROS

"In this era of high-tech toys, a sound designer has many techniques and tools to create a soundtrack. One process for creating effects is the technique labeled as "Foley." This dates back to the radio era where engineers would, in real time, create effects while actors were performing radio dramas. This system, made famous by Jack Foley himself, is basically the technique of creating a room free of noise and air conditioning as well as hum and buzz. Then, using simulated surfaces for footsteps and props for effects, sounds can be created and recorded in the controlled environment. Persons who do this day in and day out are referred to as Foley artists. In traditional film and TV, a Foley artist is required to watch, step, and create effects within two to six frames of synch, making it brilliant in the first take. For a game sound designer, a Foley stage is a plethora of opportunities for creativity. The basic rule to sound design is that there are no rules."

— Scott Gershin, sound designer, creative director, SounDeluxe

"With voice recording, I always start new actors with a few minutes of grunts and "ughs" to help break the ice. After the session is done, I get a few minutes of screams and pain sounds. If they are willing, it makes for great freebie stuff for my sound library.

"Also, really watch the levels. Clipping sounds in a game are bad. If an effect sounds bigger cut-off, make sure to pull it down and normalize."

— Christopher Roby, game/sound designer, Sinister Games

"Looping a sound is a challenging task that amounts to an art form within itself. When working on looped material such as vehicles or machinery, I like to copy the file several times and apply different amounts of compression and signal processing to each copy before looping the samples. I then find loop points on each file and listen for the best version to sit in a mix."

— Mark Showers, sound designer, Soundwerx Productions

controlled boosting or cutting around a specific frequency.

Using an equalizer effectively takes a little practice, but just as with any other effect you should have a clear idea about what you are trying to achieve before starting. Equalizers are very good at quickly boosting or cutting the bass (less than 250Hz), mid-range (500Hz to 2kHz), or high frequencies (4kHz and over) in a sound. Simulating phone lines or reduced radio transmissions can also be done by peaking between 400Hz and 3kHz. Parametric equalizers in particular are useful for eliminating unwanted noises, such as tape hiss or electric hums, at a specific frequency.

Besides equalization, audio compression is perhaps the most common method used to accentuate recordings by providing direct control over maximizing and minimizing the dynamic range of a signal. Simply stated, compression alters a signal by taking the input volume and moving it down by a factor denoted by a ratio setting. A threshold control makes the ratio change specifically at a point within the dynamic range, leaving everything below it unmodified (see Figure 11). The number of threshold changes and how dramatic they are is referred to as a "knee" setting. Hard knees cause abrupt changes in the compression ratio, while soft knees are small changes in compression staged over a range in the signal. Additionally, how aggressive the compressor acts on the signal as it's processed can be set with attack and decay parameters. Also, a compressor can be output-gained to increase the volume across all parts of the signal.

Compression tools — particularly ones in hardware — can be daunting, but slowly working through the parameters and making small adjustments while listening to the sound will make the settings much clearer. To start with sound effects, try using compression to simply regulate the high end and eliminate any distorting clip-outs. Settings with a high ratio around 3:1, a high threshold around 12 decibels, and a tight but soft knee are a good base. Bringing the knee up can provide a cleaner round-off, while bringing the overall ratio down will compress the dynamic range of the whole signal. Variations on compression include limiting

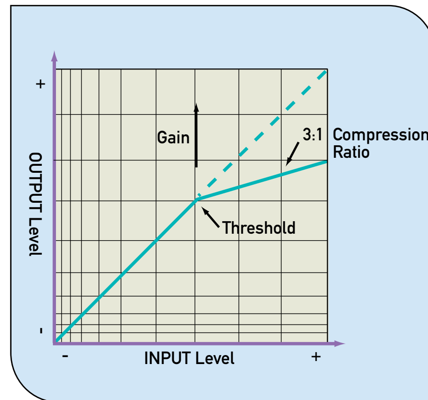


FIGURE 11. Manipulating the compression threshold at specific points within the dynamic range produces different results.

(cutting off signals above a certain level) and expanding (increasing a signal's dynamic range by decreasing low-intensity portions below the threshold). Expanding is also a good way to do smoothly transitioned noise reduction.

Pitch-shifting is another common way to improve the tonality of a sound or occasionally just to warp it into something entirely different. Subtle shifting can also be used to fix the timing on sound, for example, to synch it with an in-game animation. With a little practice, even a convincing Doppler effect (the sensation that a rapidly moving sound source, such as a car in a racing game, has a higher pitch when closing on a point in space than it does when moving away) can be created.

Modifying a sound up or down and possibly locking its duration is easily handled with pitch-transposition tools designed specifically for the task. Shifting upwards increases the frequency, making the sound higher. Conversely, pitching down provides a more low-end, basslike effect. Playing with alternating pitch (vibrato), can provide interesting results. You can also make a particular effect seem wider in a stereo mix; shifting one channel a small amount (generally less than 5 percent) will often work.

Pitch-shifting is not without adverse effects. Because sound samples are either being duplicated or removed during pitch-shifting, it's a destructive process, making it hard to backtrack by simply negating the effect. Additionally, while shifting technology has evolved to a level where interpolation artifacts are generally low or antialiasable (minimized through weighted averages), they will creep in, especially as the sound gets increasingly divergent from its original state. To be safe, pay close attention to the quality of your output, and only pitch-shift a few octaves in either direction.

Effects recorded in the studio often don't feel right when played back in the scene for which they are intended. They feel too close or too confined, like they are playing back just inches from your ears. This is where reverb comes in. The irony is that after you've spent time and resources building the perfect sound studio to eliminate reflections, now you actually want to add some. Reverb is also good for blending edits in multi-part sounds, helping them seem to be from the same acoustical page. Adding reverb is usually a fairly painless task, as most software and hardware units will have decent presets, usually named after materials or architectural spaces. Understanding what is actually happening during the process can help you engage the problem more accurately.

Reverb occurs because sound-reflecting surfaces in a space encourage the sound to bounce around, sometimes half a dozen times before finally reaching your ears. Because some sound waves will invariably travel straight to your eardrum while others take time bouncing about, you will experience the primary sound wave followed by additional, sometimes weaker, echoes of the original sound delayed by just milliseconds. The reflections in a naturally reverberant environment, sometimes dubbed a "wet" sound, will vary not only in timing but also in tonality, based on the shape, area, and absorption qualities of the surface materials in the space. Softer materials provide low-end reflections, while stiff materials bounce high frequencies.

To add reverb, start from the top by focusing on the length and density of the reflections. Working with the intensity decay rate of the reflections is also important in defining the feel of a sound's space. Short decays will suggest small spaces, while slightly longer decays will suggest much larger areas. Nonlinear falloff can suggest an unevenly occluded environment. The frequency range of the reflections is also crucial. Adding more high or low end

to a sound's reflections will cause the surface materials present in a sound's synthetic environment to feel significantly different. Be careful about overdoing it, though. Reverb is an irreversible process because editing it out is practically impossible.

In general, following the ordering of the postproduction effects in this article will have your sound ready for any last-minute finalizing and mix-down. While some steps can be skipped, it's a good idea to work at least with the compression and equalization on each waveform. Although it may work otherwise for some productions, it's usually better to apply any pitch-shifting and equalization before compression to give each process the greatest dynamic range within which to work.

While most studio-recorded effects are one-shot affairs, occasionally they will need to loop during playback. There is often little you can do to create a looping track during recording other than trying for clean, editable breaks in a sound's cycle. During editing, cross-fading multiple tracks of the same sound for loop effects that can't simply be cut in or spliced is a good starting point. For sounds that vary in timbre across their duration, another technique is to reverse a copy and then play them back-to-back.

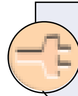
The last part of the sound design process is mixing down and doing any necessary format conversion. This can be a complex task for cutscene productions, taking days, if not weeks, in front of a nonlinear editor. However, for most in-game sounds, it's just a matter of combining any multi-part sounds into one singular effect. Keep in mind the final output during the mix-down process. If you have been working with stereo material, verify its mono compatibility. Also keep an eye on the level meters and watch for distortion, particularly if you are additively mixing material with a lot of dynamic range. If you need to down-sample your sound for output, maximize your signal-to-noise ratio with gating or compression before doing the conversion.

Wrap-Up

There is no single correct way to finalize a sound. In a game studio, it's often best to take a break when you think you've got an effect sequence done, then come back and listen to it later. Closing your eyes during playback so you can focus on the quality of the sound is also a good technique. However, testing to make sure it feels right in the game itself is ultimately the best way to make sure it really has been done satisfactorily.

Sound design is more of an art than a science. Recording and editing fresh material for your game project extends that art even further, providing an almost limitless canvas of aural creativity. But by applying knowledge from practiced sound designers with a bit of physics and a lot of imagination, the entire sound environment of your game can be raised to the next level.

Be sure to check out the expanded version of this article on Gamasutra.com.



Discuss this article on Gamastura.com!
www.gamasutra.com/discuss/gdmag

FOR MORE INFORMATION

WEB SITES

Gamasutra
www.gamasutra.com/features/index_sound_and_music.htm

Equipment Emporium
www.equipmentemporium.com/articles.htm

Acoustics & Vibrations Virtual Library
www.ecgcorp.com/velav

BOOKS

Ballou, Glen M., ed. *Handbook for Sound Engineers*, 2nd ed. Carmel, Ind.: SAMS/Macmillan, 1991.

Eargle, J. *The Microphone Handbook.* Plainview, N.Y.: Elar Publishing, 1981.

Mott, Robert L. *Radio Sound Effects.* Jefferson, N.C.: McFarland & Co., 1993.

Pohlmann, Ken C. *Principles of Digital Audio*, 3rd ed. New York: McGraw-Hill, 1995.

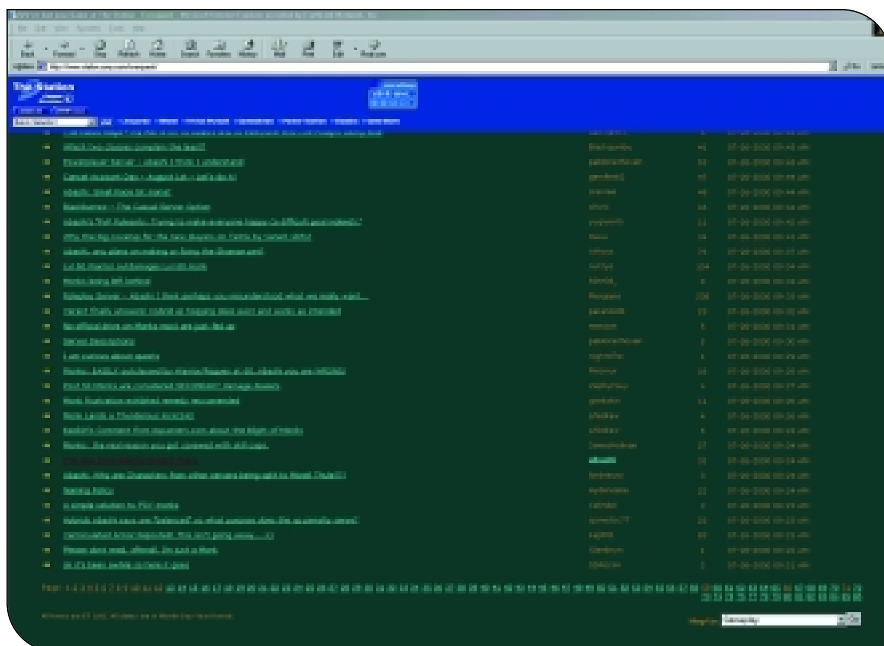
Yewdall, David Lewis. *The Practical Art of Motion Picture Sound.* Boston: Focal Press, 1999.

Taming the Customer Service Dragon in Persistent-State Worlds

It all started innocently enough. A monk had gone a-questing for the Robe of the Lost Circle, an artifact found in the massively multiplayer online role-playing game EVERQUEST.

He had spent nearly a hundred hours collecting the necessary components to make the robe, traveling all across the lands of Norrath, speaking to non-player characters (NPCs), waiting hours at a time for a particular monster to appear so he could slay it and collect an item he needed from its corpse. His path had been arduous, but the moment of truth had finally come. Breathless with anticipation, the player placed the treasures he had so carefully collected into a special container, pressed the Combine button to mold them together, and...destroyed everything by failing a skill check.

Surely this had to be a mistake, he thought. Surely the effort involved in collecting these items was the only risk the designers had intended for me to take. Had they accidentally introduced a skill check where there should be none? Upset and confused, the player posted his tale of woe on Sony's EVERQUEST boards, and rallied the support of his fellow players behind him.



The EVERQUEST message boards: 107 pages of topics, and one overworked customer service spokesman at the center of it all.

Shortly after the message board thread started, Verant's community relations spokesman, Abashi, made a post in which he stated the skill check sounded like a poor design decision, and that he would ask to see if it could be removed from the quest. Cheers rang out from the message board crowd at this display of responsible customer service. Shortly afterward, the player posted that he had requested and been given a Robe of the Lost Circle by a Game Master, one of EVERQUEST's in-game technical support personnel.

Cheers again erupted from the player's supporters, but the mood soon turned sour. The Game Master who had given the monk his robe learned the loss was due not to a software bug but that it was the intended design, and promptly took the robe away. A firestorm ensued on the EVERQUEST message boards, prompting a return visit by Abashi to explain Verant's decision: although Verant sympathized with the monk's loss, it would not be correct to overturn what had happened in the game, because the software had functioned as designed.

To Be... Or Not to Be?

So what should have happened? Was it reasonable for the player to ask for a new robe after Abashi said the skill check was a poor design decision, and would likely be removed? Should the Game Master have given the player a robe when he thought the loss was due to a bug? Was the decision to take back the robe correct? Should Verant have overturned that decision, because the robe

DEREK SANDERSON | *Derek is a systems designer for ULTIMA WORLDS ONLINE: ORIGIN. He has held a variety of design, marketing, and customer service positions in the persistent-state world market over the past few years, and tries to use that experience to build "zero support call" systems for his games. Contact him at gamedesigner@aol.com.*

had already been given to the player for a problem that was going to be fixed anyway? It's a tricky issue, certainly, but a typical one for persistent-state world (PSW) games such as EVERQUEST. The answer is even trickier. All four of the decisions were correct, and they perfectly illustrate the PSW customer service paradox: Sometimes you have to refuse an individual customer's request in the interest of maintaining a balanced game.

Obviously this creates a conflict of interest for game administrators. On one hand, we want our customers to be happy; happy customers are loyal customers. On the other hand, our customers pay us to referee the game world for them, which means we must keep that world fair and balanced, making the risks and rewards equal for all. Sometimes we will design those risks and rewards poorly, and one of our players will have an unpleasant experience because of it, such as our friend the monk. But if we change the outcome for him, are we not giving him a competitive advantage over every other player who accepts the negative consequences our game imposes?

Refusing any customer is a no-win situation. Customers ask for our help when they feel wronged, and do not care about lofty philosophies such as the balance of the game. They simply want their concerns addressed. We may not be able to help them with every individual problem, but a good PSW designer will watch for patterns of complaints and strive to adjust the game systems so they do not generate customer service calls. This article will describe a few rules common to creating that mythical "zero support call" game.

Some Ground Rules

Let's begin with the cardinal rule of online customer service, which is:

If you don't like what is happening in your game, you should first look for a design solution rather than have your customer service staff implement ad hoc fixes. You'll save time, money, and staff morale in the long run.

A company that relies on its customer service staff to fix problems generated by its own game systems is one that will quickly see its customer service calls spiral out of control as its player base starts depending on them to fix every minor issue that arises. You're only fighting yourself if you do this, and you should therefore only use ad hoc adjustments to game mechanics on a temporary basis when you fully intend to change the system that generates the calls in the first place.

I can best illustrate this with a story from my early design days working on Simutronics' venerable text MUDs, GEM STONE III and DRAGON REALMS. Both games — still going strong at www.play.net — had their roots in the old Genie BBS service run by General Electric, and their design and customer service policies reflected their small, pay-by-the-hour community with a high staff-to-customer ratio. That community's bonds were strained almost to the breaking point when the games went from 50 simultaneous customers on Genie to hundreds on AOL in late 1995, then thousands a year later when AOL dropped its hourly charges in favor of a flat $20 monthly fee.

The biggest difficulties resulted from the loss of accountability for one's actions that had been present in Genie's small community. Many antisocial activities, though allowed by the game mechanics, simply did not happen on Genie because the players would exact swift retribution upon anyone who violated the community's mores. For example, in the Genie days, if someone died and left his equipment on the ground, other players helped him get his items back. Stealing from the dead was almost unheard of, and the few incidents that occurred caused such an uproar among the players that they were talked about for months afterward.

Imagine the surprise of the Genie players when America Online came, bringing with it the anonymity of five screen names per account, each of them capable of playing the game with a new character, and no way to tell which characters were related to each other. This anonymity, just as it does in all other areas of the Internet,

bred disruption: within weeks we were besieged by anonymous characters who logged in to rob dead adventurers, only to run away, hide, and then log off, never to be seen again.

Because our games had a high level of item persistence — items rarely broke or wore out — players who had been stolen from in this manner not only felt violated, but often lost weeks or even months of work from a single theft. And since the player not only didn't know who the real thief was, but also had no way of retrieving his items from someone who had logged off immediately afterward, an incident of such looting inevitably generated a furious customer assistance call demanding we do something about the problem.

At first, we tried an administrative solution. We considered the looting to be cheating, both because it involved the use of an alternate, anonymous character to hide the looter's identity, and because the looting character inevitably logged off immediately after the theft, abusing the game's logoff mechanics to avoid retribution. Because it was cheating, the looting was an invalid action, and we confiscated the items and issued a warning to the offending player for the disruption.

Our intentions were good, but we were fighting a losing battle. After all, having multiple screen names (and therefore multiple characters) was an integral part of AOL's system. Our game mechanics permitted dragging dead bodies to out-of-the-way places. Those same game mechanics forced dead bodies to decay, leaving all the items on the ground free for the taking a few minutes after death. Instant logoff was also part of the game. Thus, it was inevitable that players would use these mechanics to their advantage, whether it was playing fairly or not.

Trying to control the problem with policy was expensive. The more often we intervened, the more our customers expected us to do so. We still had to upgrade our games and provide normal customer service, and every hour spent tracking down an anonymous troublemaker meant one less hour devoted to development. We found ourselves spending more and more time hunting down thieves and returning items, so much so that some of our customer service staff did little else.

We quickly learned we would not be able to stem the tide without significant design changes, however, and struck back with a variety of systems that restricted players' ability to harass dead adventurers. One of those systems, for example, prevented a player from moving another's corpse without the dead player's permission. Another allowed players to protect their corpses from being looted for a few minutes via a protective spell. A third was what we called our "Graverobber" system.

The Graverobber system tracked the owner of any item left on the ground at death. If another player picked up the item, the system tracked that person and anyone else to whom he subsequently passed the item. If someone holding a stolen item logged off or lost connection within a few minutes after the theft, the system removed it from the player's inventory and sent it back to the room from which it had initially been taken. Under optimal circumstances, the system worked beautifully: if an anonymous character stole something and logged off, the item would return to the owner's body, and the staff would not have to intervene.

Circumstances were not always optimal, however, and though the system stopped many "loot and scoot" problems, we sometimes found our own code working against us, since the Graverobber system did not differentiate between friend and foe. Players frequently picked up items for a dead friend, then lost connection while traveling back to a safe area, only to have the items returned to the room in which their friend had died and be subsequently

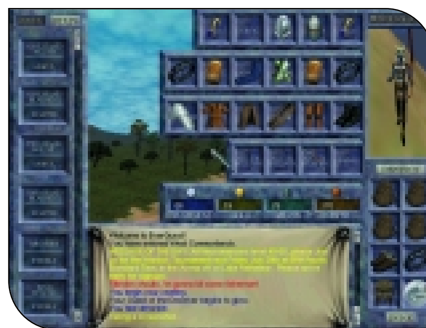


FIGURE 1. EVERQUEST's inventory screen, overlaying the game window. If a player misses while placing an item into the lower-left inventory slot, it will be dropped into the world, perhaps never to be recovered.

deleted by our garbage collection system because no one was around to pick them up. Those who lost items in this manner inevitably asked us to replace them, and we inevitably refused — after all, the system was functioning exactly as we had designed it, wasn't it?

As you may have guessed, this was not an answer our customers were pleased to hear, any more than our friend the EVERQUEST monk was pleased to hear his robe would not be replaced. This is because:

Your customers expect your world to be predictable, just as they expect the rules of any game to be predictable. They will view unpredictability as a flaw, and expect you to repair the consequences of that flaw.

DRAGON REALMS customers who lost items to the Graverobber code were upset because the code was intended to stop looting, not delete items because a friend had inadvertently logged off before an arbitrary timer had expired. Although the code was functioning exactly as it was implemented (much like the skill check for the monk robe), it was not functioning as it was intended. Intent is predictable. In a complex world simulation, however, individual game systems often combine to cause unpredictable side effects in their actual implementation.

"That's Just How Things Are"

Another example from EVERQUEST is apropos here. A writer from the web site Gamefan was visiting one of EVERQUEST's islands one evening last June when he was beset by a pirate, one of the game's huntable creatures. Not having his weapons ready, the writer first beat a tactical retreat until he was out of range, then attempted to equip his sword to prepare for battle. In his panic, however, he accidentally clicked on the game window with sword in hand, which caused it to drop into the world rather than his readied weapon slot.

This problem happens occasionally in EVERQUEST because of the way its inventory interface is designed, which you can see in Figure 1. Each piece of equipment a character wears is assigned a particular inventory slot, with said slots being dis-

```
Microsoft(R) Windows 98
   (C)Copyright Microsoft Corp 1981-1999.

C:\WIN98>tracert simutronics.com

Tracing route to simutronics.com [198.83.204.1]
over a maximum of 30 hops:

  1   105 ms    97 ms    98 ms  38.1.1.1
  2    95 ms    95 ms   110 ms  austin2.tx.pop.psi.net [38.146.200.1]
  3   123 ms   121 ms   104 ms  38.1.42.7
  4   115 ms   122 ms   108 ms  38.1.22.193
  5   136 ms   135 ms   137 ms  38.1.10.80
  6   155 ms   156 ms   163 ms  Serial3-1.BR2.CHI1.ALTER.NET [137.39.250.37]
  7   153 ms   157 ms   159 ms  137.39.23.34
  8   170 ms   171 ms   286 ms  s6-0-0.t80-1.St-Louis.t3.ans.net [140.223.25.14]

  9  1021 ms   170 ms   185 ms  f0-0.c80-10.St-Louis.t3.ans.net [140.222.80.198]

 10   154 ms   150 ms   157 ms  h0-0.enss3139.t3.ans.net [207.26.58.214]
```

FIGURE 2. The Tracert program might show your customers that a latency problem is not your doing, but they still will expect you to correct the consequences of that latency.

played directly over the world window when a character manipulates his inventory. Click a few pixels too far to the left and the system assumes you want to drop an item into the world, rather than your inventory. This is normally a recoverable mistake, as dropped items generally persist for a few minutes before the system deletes them. The player knew this and turned to pick up his sword, when, according to him, it disappeared. Frustrated, he called a Game Master, explained what had happened, and was allegedly refused a replacement because there was no proof he had lost the weapon to a bug. Outraged at this perceived injustice, the Gamefan editor went on to publish an angry editorial about the incident and other examples of what he perceived to be Verant's poor customer service.

Now, I would argue the issue was not one of poor service so much as another illustration of the problems that ensue when implementation and intent conflict. Verant designed an interface that made it possible to drop an item into the game world — though there is rarely a need to do so — and also designed a garbage collection system that eventually deletes dropped items so the world does not fill with clutter. They obviously did not intend these two systems to function as a way to delete items from their players' inventory, so players who lose items in this manner inevitably request a replacement.

Verant's Game Master was unwilling to replace the sword, however, because the loss did not occur due to a genuine bug but rather through player error. In fact, from the eyes of a game provider, the event was an entirely predictable one: if you're careless when placing items into your inventory slots, you will throw the item into the game world, where it is very likely to be removed by the garbage collection system. Players are meant to understand that this is "just how things are" in the game, and to be careful when placing items into their inventory, hence they shouldn't expect the game administrators to make up for their mistakes. Problem solved, right? Well, no, not really, because players don't automatically understand the designers' notion of "that's just how things are."

"That's just how things are" describes predictability on a game-wide, "macro" level. Such predictability is worthless from a customer service perspective. Only individual, "micro" predictability will make your customers happy.

On a macro level, about 50,000 Americans will die in car wrecks every year. That's just how driving is. On a macro level, roughly one out of every 150,000 jetliner flights will crash. That's just how flying is. On a macro level, people will get mugged, struck by lightning, or have their homes destroyed by natural disasters. Those things happen to people who are in the places where they occur, so the victims should just accept their fate and move on, right? Nope. Macro predictability does not make us happy in real life, so why should we expect it to satisfy our customers inside our world simulations?

This issue affects not only your game systems, but also every other aspect of your customers' play experience, even their connectivity issues. Your customers, for example, will behave as if they do not believe in the Internet. A customer's modem drops its connection? Your fault. A farmer in Pig's Knuckle, Arkansas, cuts a major communications line, causing extreme latency not only in your game, but across the entire eastern seaboard? Your fault. Terrorists from a rival game company blow up a router down the street? That's your fault, too. And because it's your fault, your customers will expect you to fix anything that adversely affects their play experience as a result. Once again, the issue is one of differing expectations. As game designers, we are well aware of the issues the Internet has as a gaming platform. It does things that hamper the usual flow of gameplay in our worlds. On a macro level, we know that while we can't predict each occurrence with pinpoint accuracy, over the course of a month's playing time a certain percentage of customers will experience high latency, packet loss, or dropped connections. That's just how things are on the Internet.

However, the player does not experience your game from your perspective; he experiences it as an individual. To him, these events are exceptions to the game environment you have created, not part of the normal "rules." The player would not intentionally make his character kill himself by leaping off a cliff, so why should he do so just because he lost connection while running? The player would not make his character suddenly stop fighting in the middle of a battle with a dragon, so why should he stop because the server is not receiving his "fight" packets? Players want to be in full control of their characters when something bad happens to them, and if they're not, they will demand that you rectify any adverse consequences they suffer, even if you can show them that the real problem is not the game itself (Figure 2). Telling them, "Occasionally losing your

connection is just part of playing games on the Internet," is like telling someone, "Having your little brother kick over your Monopoly board is part of playing Monopoly in his room. Sorry, I won't replace your hotels." Sure, misbehavior from nasty kid brothers is something that can be predicted, but they are not part of the rules of Monopoly. Therefore:

Good disaster recovery mechanisms are vital to a successful PSW. Your volume of customer service calls is inversely related to the ease with which players can prevent, and correct, their mistakes.

Making it easy for players to fix their own mistakes is the equivalent of placing a save-game feature into an online environment. This is a difficult thing to accomplish efficiently in PSW design, as there is a fine balance between allowing players to correct mistakes and maintaining challenge in the game. The trick is to pinpoint those areas where your players most frequently ask you for assistance, and enable them to accomplish these things themselves.

Getting back to the above example, in which hordes of hapless players go plunging off cliffs every time their packets miss that left turn at Albuquerque, one way to reduce calls demanding reparations would to be make someone stop running immediately should your system detect a loss of connectivity. Another would be to restrict the amount of damage one can take from falling, so it could injure but never kill. A third would be to make characters stop at the edge of any slope that could cause falling damage, and require a separate keystroke before leaping. You should even go so far as to question the very existence of any system that generates multiple support calls — is falling damage really necessary in your game? Sure, it's realistic, but unless it's needed for some vital game system, it probably doesn't add a lot of fun to your gameplay.

In the aftermath of the Gamefan editorial, many players suggested Verant add a confirmation dialog that had to be closed whenever a player attempted to drop an item into the game window. The suggestion has a precedent in EVERQUEST; a similar dialog (see Figure 3) pops up whenever a player attempts to destroy an item, and if players find the confirmation dialog annoying, they can turn it off in an options menu. To Verant's credit, they saw the value in these suggestions, and recently implemented just such a change. Players may now set their client to demand a confirmation before dropping items into the world, thus resolving the conflict between design and intent, and thereby probably reducing their volume of customer service calls.

FIGURE 3 (above). EVERQUEST requires that users confirm their decision before permanently destroying an item.

A Benevolent Dictatorship

Before you run off to make exciting new changes to your game, however, keep in mind that anything you do will have a very real impact on the lives of your customers. You may have designed the game, and you may have a vision for what it should be like now and in the future, but if you expect a high degree of customer loyalty you must encourage your players to believe they are full citizens of your world. They are not simply purchasing a service from you, and are not simply playing a game. Players live in the world you create, and any changes you make affect their daily lives almost as much (and more so in some cases) as anything that happens in the "real" world.

Let your customers know what you are thinking, and ask for their input, before making even the smallest of changes to your game mechanics.

Of course, PSW administration can't be a democracy. Sometimes hard decisions must be made that your entire player base opposes, such as reducing the power of a popular spell or item in order to eliminate an exploit. However, if you explain to your customers your reasoning for doing so in advance, you will gain much greater acceptance of your changes than if you simply presented them with a *fait accompli*. The feedback you receive may even present you with better alternatives you hadn't considered. After all, the minds of your customers are legion, and you are simply a small group of designers who like to make computer games. At Origin, one way we like to keep in touch with customers is with a "Question of the Week" forum (Figure 4), in which we bounce ideas off future players of ULTIMA WORLDS ONLINE: ORIGIN.

You need to remember that your customer base includes a wide variety of people; be careful not to fall into the trap of only talking to your "elite." Your design team may be full of role-players, but the mass market most certainly is not. Your design team may post on popular fan web sites, but again, the mass market does not, nor do they participate in IRC chats, nor do they write you letters telling you how they feel about the game. The mass market is a great, silent majority that you must reach out to in order to learn what they really want. Don't fall into the trap of assuming everyone is like you, or that the communications that fall in your lap are representative of your entire customer base.

A final word of caution:

Your customers will overwhelm traditional channels of communication, and will become angry when you cannot keep up with the demand.

Nothing frustrates your customers more than being given an expectation of customer support that isn't met. Yes, many customers will demand far more individual attention than is reasonable for ten dollars a month, but it absolutely infuriates them when a company provides, and promotes, a channel of communication and then does not respond promptly to requests sent through that channel. Falling behind only exacerbates the problem; if you don't answer their phone calls, they'll send you angry e-mail. If you don't answer their e-mail promptly, they'll send you more e-mail demanding to know why you haven't answered their e-mail. Don't answer those e-mails, and they will flame you on the message boards.

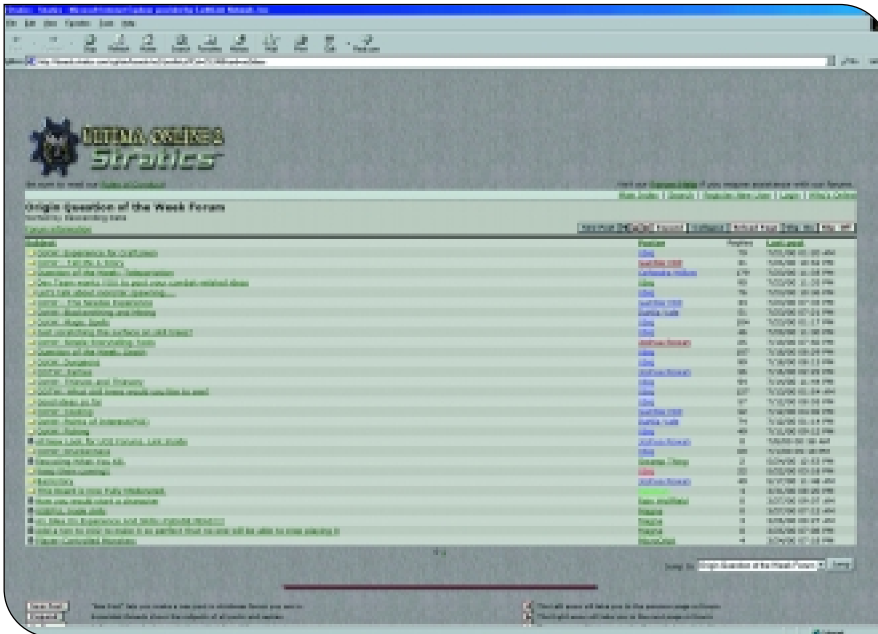


FIGURE 4 (top). The ULTIMA WORLDS ONLINE: ORIGIN staff discusses design issues with their future customers in a "Question of the Week" forum.
BOTTOM, ULTIMA ONLINE's Help screen is an example of a system designed to decrease the customer service load by prescreening assistance calls and directing them to the best forum for getting problems resolved.

Don't respond on the message boards, and they will page your in-game staff to gripe about your poor phone and e-mail service, over which your in-game staff has no control. This makes them think your in-game staff is ignoring their concerns, and they will call your main offices to gripe about your staff ad infinitum.

The solution is not to throw more bodies into your customer service department, as there is an insatiable demand for individual attention in your customer base that is impossible to satisfy if you expect to make a profit. You must instead be just as clever designing your customer service systems as you are in designing your game, because your customers will get just as frustrated over your customer service when its intent (does this sound familiar?) does not match its implementation.

No matter how you structure your service, be sure your representatives have the ear of your designers. The people who deal with the customers every day are the ones who know best what aspects of your game design are generating the most problems, and if you let them be a part of solving those problems, they'll be much happier and more productive in the long run.

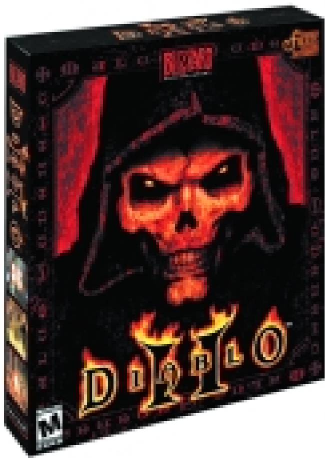
Design Is Hard. Customer Service Is Hard. Writing Articles Is Easy.

I've used examples mainly from my competitors' games, but I want to make it clear that the problems I have described are a perfectly normal part of designing and administrating any PSW. Hindsight is 20/20, and it's easy for me to sit in my ivory tower and say, "You should have done this," and "You should have done that." But even the best designer can't predict everything that could go awry in a game before it is released, and even the best customer service department can't help but feel overwhelmed at the never-ending demand for personal attention its customers expect.

If you remember nothing else from this article, keep this bit of advice: Remind yourself every day that you are creating worlds not for yourself, not for your company, but for thousands of individuals who make individual decisions every day about whether to continue playing your game. Listen carefully when they tell you something is making them unhappy, and though you must be a dictator, be a wise and benevolent one. Give freely to your subjects whenever possible. Fix the little things as well as the big, because easy changes are appreciated just as much as the hard ones. And above all, have fun.

Discuss this article on Gamastura.com!
www.gamasutra.com/discuss/gdmag

Blizzard Entertainment's DIABLO II



GAME DATA

PUBLISHER: Blizzard Entertainment

NUMBER OF FULL-TIME DEVELOPERS: 40

LENGTH OF DEVELOPMENT: more than 3 years

RELEASE DATE: June 28, 2000

PLATFORMS: PC and Macintosh

DEVELOPMENT HARDWARE USED: Typical programmer workstation: 500 MHz Pentium II running Windows NT with 128MB RAM and 9GB hard drive. Typical artist workstation: dual 500 MHz Pentium IIs running Windows NT with 256MB RAM and 14GB hard drive.

DEVELOPMENT SOFTWARE USED: 3D Studio Max, Photoshop, Microsoft Developer Studio/Visual Studio and SourceSafe

NOTABLE TECHNOLOGIES USED: Glide, Direct3D, RAD Game Tools' Bink, DirectSound3D, and Creative Labs' EAX.

The original DIABLO went gold on the day after Christmas in 1996, after a grueling four-month crunch period. We hadn't put any thought into what game to do next, but as most developers can probably relate to, we were pretty certain we weren't ready to return to the DIABLO world after such a long development cycle. The only thing we were certain of was that we wanted to avoid another crunch like we had just experienced. DIABLO II went gold on June 15, 2000, after a grueling 12-month crunch period.

After DIABLO shipped, we spent about three months recovering and kicking around game ideas for our next project, but nothing really stuck. The idea of returning to DIABLO began to creep into the discussions, and after a couple of months of recuperation, we suddenly realized we weren't burned out on DIABLO anymore. We dusted off the reams of wish-list items we had remaining from the original, compiled criticisms from reviews and customers, and began brainstorming how we could make DIABLO II bigger and better in every way.

DIABLO II never had an official, complete design document. Of course, we had a rough plan, but for the most part we just started off making up new stuff: four towns instead of the original game's one; five character classes, all different from the previous three; and many new dungeons, vast wilderness tile-sets, and greatly expanded lists of items, magic, and skills. We wanted to improve upon every aspect of the original. Where DIABLO had three different armor "looks" for each character, DIABLO II would use a component system to generate hundreds of variations. Where DIABLO had "unique" boss monsters with special abilities, DIABLO II would have a system for randomly generating thousands of them. We would improve the graphics with true transparency, colored light sources, and a quasi-3D perspective mode. Level loads would be a thing of the past. The story would be factored in from the beginning and actually have some bearing on the quests. We knew creating this opus would be a big job. Because we had the gameplay basics already polished, we figured we would hire some new employees, create some good tools, and essentially make four times the original game doing only two times the work. We estimated a two-year development schedule.

The DIABLO II team comprised three main groups: programming, character art (everything that moves), and background art (everything that doesn't move), with roughly a dozen members each. Design was a largely open process, with members of all teams contributing. Blizzard Irvine helped out with network code and Battle.net support. The Blizzard film department (also in Irvine) contributed the cinematic sequences that bracket each of DIABLO's acts, and collaborated on the story line.

Almost all of DIABLO II's in-game and cinematic art was constructed and rendered in 3D Studio Max, while textures and 2D interface elements were created primarily with Photoshop. The programmers wrote in C and some C++, using Visual Studio and SourceSafe for version control.

Blizzard North started out as Condor Games in September 1993. The first contracts we landed were ports of Acclaim's QUARTERBACK CLUB football games for handheld systems and, more significantly, a Sega Genesis version of JUSTICE LEAGUE TASK FORCE for Sunsoft. Silicon and Synapse, a developer that would later change its name to Blizzard Entertainment, was developing a Super Nintendo version

ERICH SCHAEFER | *Erich is vice president of Blizzard North and one of its founders. Erich played a leadership role in the management, design, and art direction of DIABLO and DIABLO II. He got into the game development business from a background of graphic design and goofing off. With DIABLO II on the shelves, Erich and his new bride Hanna finally have time to get the house in order.*

While the player characters are only seen in the game as 75 pixels tall, all were modeled and rendered in high resolution for use on the character selection screen and in promotional materials. Here, the Paladin stands tall.

Creating detailed sketches of settings, such as this hut in the Act III dock town of Kurast, preceded the actual modeling of background art.

Much time was spent perfecting Act I since it would likely be used in a beta test or demo. The Amazon was the first character to be completed.

of JUSTICE LEAGUE TASK FORCE. Condor ended up pitching the idea for DIABLO to Blizzard, and halfway through the resulting development process Blizzard's parent company acquired Condor, renaming us Blizzard North. Throughout a tangled history of corporate juggling and ownership changes, Blizzard North has remained a very independent group. Our staff has grown steadily from about 12 at the start of DIABLO to 24 at the start of DIABLO II, and finally to our current group of more than 40. We concentrate 100 percent of our efforts on game development. To help keep this focus, Blizzard's headquarters in Irvine manages other functions, such as quality assurance, marketing, public relations, technical and customer support, as well as the operation of the Battle.net servers. Our parent company, Havas Interactive, deals with business functions such as sales, manufacturing, and accounting.

What Went Right

1. DIABLO II is still DIABLO. A constant theme in previews and reviews of DIABLO II was that we didn't change anything; it was more of the same. At first that struck us as odd. We kept less than one percent of the code and art from the first game. We rewrote the graphics engine, changed all the character classes and skills, shifted and expanded the setting, reworked and added to the magic items, brought back only a handful of our favorite monsters, and designed a ton of new gameplay elements, such as running, hirelings, left-click skills, and random unique monsters. Why, then, did everyone think it was the same thing? In the end, we decided just to take it as a compliment. The play-testers and reviewers meant they were having exactly the same kind of fun that they had in the original game.

Both DIABLO and DIABLO II provide a constant source of simple pleasures, many of which are perhaps too basic and obvious to mention in evaluations and reviews, but which are fundamental to their success. We used the term "kill/reward" to describe our basic gameplay. Players continually kill monsters and get rewarded with treasure and experience. But the rewards don't stop there. We offer a steady stream of goals and accomplishments to entice the player to keep playing. There's always a quest that is almost finished, a waypoint almost reached, an experience level almost achieved, and a dungeon nearly cleared out. On a smaller scale, we tried to make every single action fun. Moving around inventory items produces pleasing sounds. Monsters die in spectacular fashion, like piñatas exploding in a shower of goodies. We strove for overkill in this sense, in that players are constantly on the verge of something great — only a few mouse-clicks away from a dozen interesting things.

DIABLO II retained DIABLO's randomly generated levels, monsters, and treasure. This obviously allows for better replay potential, but also serves to make each player's game his or her own. Players feel an ownership of their own game experience in that they are actively generating a unique story. It's enjoyable to tell friends about what you have just done in the game, knowing for sure that they have not done the same thing. Simply following an online walk-through won't help them accomplish goals without effort.

Finally, DIABLO and DIABLO II are easy to play. We used what we call the "Mom test": could Mom figure this out without reading a manual? If we see new players struggling with how to sell items, we look at how they're trying to do it and make that way work too. We strove to make the interface as transparent as possible. You want to open a door? Left-click on it. Want to move to a target location? Left-click on it. Want to attack a monster, pick up an item, or talk to a non-player character? Well, you get the idea. It's amazing how many games have different controls and key combination for all these actions when simpler is always better.

2. Blizzard's development process. Blizzard's development process is designed to ensure that we make a great game. While our goal is to meet the milestones we set, our process, in terms of design and business, is structured

to allow us to wait until the game is as good as it can be before we ship it. We recognize that not all developers have this same opportunity, but many of the methods we use along the way are applicable to any development environment.

First, we make the game playable as soon as possible in the development process. Our initial priority was to get a guy moving around on the screen and hacking monsters. This is what players would be doing most of the time, and it had to be fun. We were constantly able to hone the controls, pathfinding, and feedback mechanisms during the entire length of the game's development. Most importantly, it allowed us to determine what was fun to do, so we could provide more of it, and discover what was awkward or boring, so we could modify or remove it. For instance, it became obvious very early that players would be killing large amounts of the same monsters, and those monsters would predominantly be attacking the players. This gave us the opportunity to plan for multiple death sound effects and additional attacking animations for each monster. If we hadn't experienced the core gameplay as early as we did, combat would have ended up feeling much more repetitive.

Also, we constantly reevaluate gameplay and features. Up until the very end, if we can make the game better we will, even if it means redoing big tasks. For instance, we decided that we didn't like the Bone Helmet graphics for the characters more than a year after having rendered them, but we went ahead and remade them, even though it took a couple of weeks and the collaboration of four artists. Only weeks away from scheduled beta testing, we scrapped our Act IV level layout schemes because they were just a bit too empty and similar. The last-minute fixes turned these levels into some of the best, befitting their climactic function. DIABLO II took more than 40 people and over three years, essentially because we made two or three games and pared them down to the best one.

Another gigantic reason for our success is our open development process. We strive to hire people who love games, and we make games that we want to play. Every member of the team has input into all aspects of the game. Discussions around the halls and at lunch become the big ideas that shape the game. A programmer suggested to a designer the concept of gem-socketed, upgradeable weapons, which turned out to be a huge crowd-pleaser. A musician's dislike for the old frog-demon's animation inspired us to redo it. As a team, we don't have to wonder what our audience wants, because we are our audience. If we like the game we are making — especially if, after two years of playing it, we are not bored to death — the game is clearly going to be a winner.

3. Character skill tree. Our most revolutionary new idea was the character skill tree. For a character to attain more powerful skills, he or she must master prerequisite skills. The ability for characters to branch into different areas of the skill tree, and to choose a level of specialization in each skill along the way, provides truly unique characters.

At the start of development, we planned to use the model from the original DIABLO: characters would find and read books to learn spells and skills. Unlike DIABLO, which had 28 spells shared by all three characters, we wanted to create a separate group of 16 skills for each of our five new character classes. This would definitely have been an improvement, but every character of a given class would still end up knowing all the same skills as other members of their class. Another problem was that players would likely be finding spell books for other character classes much more often than for their own. The skill tree solved these problems. The general idea was taken from the tech trees many strategy games employ. In strategy games, players advance by researching new technologies, which in turn open up further avenues of research. We adapted this to have our characters advance by choosing a new skill or strengthening an old skill every time they gain an experience level. Characters can generalize by choosing a wide variety of skills, or specialize by allocating many skill choices into a small group of skills. We also created a strategy element of choosing skills you might not use, just so you can get to one further up the tree later.

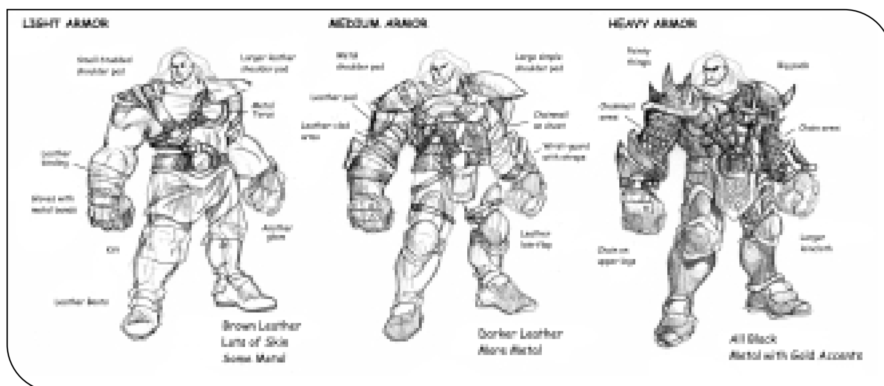
The end result of the skill tree is that one player can develop a Necromancer who kills monsters with a powerful poison dagger skill augmented by curses that cause monsters to fight each other, while his friend's Necromancer will summon hordes of skeletons to fight for him, and doesn't use any curses at all. The longevity of DIABLO II will be enhanced by the endless strategies that can be debated and experimented with.

4. Quality assurance. The task of testing a game of DIABLO II's scope, with its huge degree of randomness and its nearly infinite character skill and equipment paths, required a Herculean effort. We found we could not play-balance the climactic fight against Diablo without actually playing the entire game up to that point, because we could not predict what kinds of equipment a character might have, or what path through the skill tree he or she may have followed. This meant 20 or 30 hours of play for all the different characters, with a good variety of skill sets and equipment for each.

The architecture in DIABLO combines aspects of many different cultures in order to arrive at an interesting mix that doesn't look too much like any single one. Here, the buildings of Travical from Act III are based on Mayan and Aztec references.

Whenever we changed the game's treasure spawn rate or experience curve, we had to test it all again. Further complicating matters were multiplayer and difficulty-mode balance. Would a party of five Paladins, each using a different defensive aura, be untouchable? After more than 100 hours of play, is a fire-based Sorceress unable to continue in "Hell mode"?

The QA team created a web-based bug-reporting database through which we categorized and tracked all bugs, balance issues, and gameplay suggestions. In the end, this list delineated more than 8,300 issues and suggestions. Well-organized teams of testers concentrated on different aspects of the game, divided into groups that would specifically test character skills, item functionality, monster types, and spawn rates, or explore the countless variations found in the random level generation system. The members of the QA team became very good players and astute observers of the progress of the game. Everything worked much more smoothly than our experiences with the original DIABLO.



TOP. The player characters have modular armor of three varieties, light, medium, and heavy, which were mixed and matched to provide more individualized character appearances. "Paper dolls" created on paper and in Photoshop allowed mixing and matching of different pieces of armor to see how they worked together on the Barbarian.
BOTTOM. The Barbarian, translated from the sketches into a full, high-polygon model. Each part of a character's armor (the head, the torso, the legs, each arm, a weapon, and a shield) was rendered separately with in-house tools.

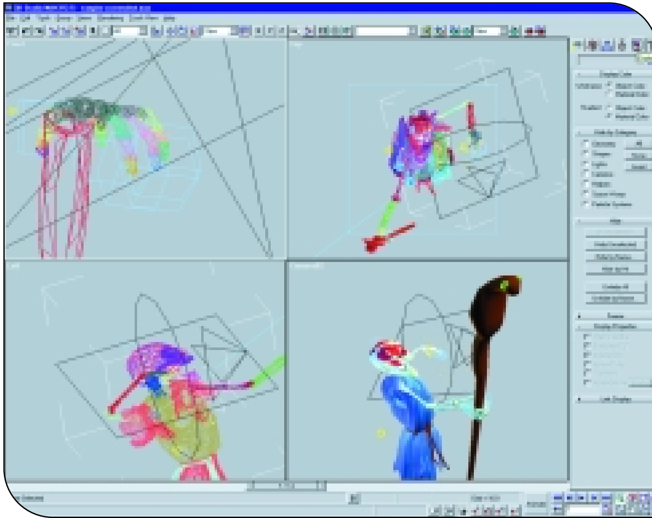
5. Simultaneous worldwide release. In the past, Blizzard's strategy for shipping its game has been to get games on North American retailers' shelves as quickly as possible after the English version of the game went gold. With the original DIABLO, we created our gold master on December 26, and some stores had it on the shelves by the 30th. Since DIABLO was released, the percentage of international customers had increased substantially, and with DIABLO II, we fully expect more than half of our sales to come from outside North America. With such a large number of customers located outside the United States, for DIABLO II we decided that there would be significant advantages to coordinating the U.S. release to coincide with the rest of the world, not only to build anticipation for the product, but for the benefit and satisfaction of our customers as well.

If we release a game in the United States first, customers in the rest of the world don't want to wait a few months while we translate and localize it for their country. Due in part to the international climate fostered by the Internet, players around the world all know about the game at the same time and want to get it while it's hot. They might buy the U.S. version under the table or search out a pirated copy. Worse, they might lose interest by the time we release a localized version. DIABLO II's simultaneous worldwide release also allowed our marketing and PR departments to focus their efforts toward creating a frenzy of interest for the first week of sales. Although the simultaneous release was a logistical headache, it was all worth it in light of DIABLO II's superb success.

What Went Wrong

1. Developing the new Battle.net. We have always been very proud that our company launched Battle.net with the original DIABLO. Just a couple of months after DIABLO shipped, Battle.net was the largest online game service in the world. At DIABLO II's launch, Battle.net had more than 6 million unique active users.

Despite the original DIABLO's success online, we knew as we began development that to create the type of multiplayer expe-



Characters and monsters, such as this Vampire, were created in 3D Studio Max. An in-house tool would render the files from many different angles (eight for all monsters, 16 for player characters), and export them in the file formats used in the game.

rience that we wanted to achieve in DIABLO II, we would need to fundamentally change the game network. And, as we expected, this became one of our biggest challenges during development. We had to reinvent Battle.net's structure by melding existing technology with new programming and feature sets. This had implications across the board. We had to rethink everything — programming, hardware, bandwidth, staffing, online support, and how we could financially support this model while keeping it free.

Although the original Battle.net had been further modified to support STARCRAFT as a chat and matchmaking service, for DIABLO II we needed much more: game servers where the Realm games would actually be played, secure character-data servers, and game tracking systems. Trying to shoehorn these elements in the existing Battle.net system proved very difficult. For instance, we planned to have character names represent players in Battle.net, but it was designed to handle chatting between account names. It took a lot of design and implementation time to arrive at our final system, where users see character names but have to send remote messages to account names.

We initially believed that working with the existing Battle.net would save us time, but in retrospect, we learned that melding technologies is a difficult process, and in some cases, recoding instead of integrating is the better course of action.

2. Launching the new Battle.net. The success of Battle.net after DIABLO's launch created a new challenge for us. When DIABLO was released, Battle.net was a new online service. Basically, we were able to ramp up as more customers joined the service. When DIABLO II shipped, Battle.net had millions of users. The level of anticipation was higher than for any of our other games. We were well aware of the expectations, and we knew that no other company had ever attempted to create and sustain an online service that could support the type of usage DIABLO II would experience right out of the chute.

We spent countless hours preparing for Battle.net's DIABLO II debut. We teamed with the best ISPs in the word, and conducted months of internal and external beta testing. We ramped up bandwidth and hardware. We beefed up the Battle.net, quality assurance, and support service teams.

Although we had more than 100,000 people testing the DIABLO II Realms, having more than one million customers in just three weeks proved to be very different from beta testing. The beta test was very successful in uncovering many stability issues that were addressed before the launch. After the game shipped, we faced bugs that only appeared at much higher usage rates. The issues that we faced at launch were ones that could not have been simulated in a beta test of 100,000 people. It took a much larger influx of players to trigger certain situations.

Knowing the massive scope of what we were trying to achieve with Battle.net, we had measures in place at launch to help us deal with issues that arose as usage increased. For instance, we maintained both a team of programmers and the entire quality assurance department to solve problems as they appeared, and had our support team working overtime. We also had an action plan in place to increase hardware and bandwidth as needed.

In some respects, we are victims of our own success. We underestimated sales, but we also underestimated the allure of playing on the Battle.net Realms. By solving the cheating problem in DIABLO and enhancing Battle.net with new features — such as the ability to see everyone's characters in the chat room — we seem to have attracted a larger share of Battle.net players than with any of our previous titles.

3. Graphics. Shortly before DIABLO II shipped, we began noticing some feedback from customers about the resolution of the graphics in the game. They were frequently labeled "outdated" or "pixellated." The shame is that the technology choices we made eclipsed the recognition of the fantastic job the artists did. We put a lot of effort into creating characters, monsters, and landscapes with a lot of

unique character. The game displays an incredible amount of action happening on-screen in an easy-to-follow manner. Still, with all the negative reaction, we probably should have done it differently.

When we began producing art for DIABLO II in mid-1997, we investigated a lot of options. We mocked up a 3D engine and checked out voxel systems. It didn't take us long to go back to what we used in DIABLO: 2D graphics at 640×480 resolution with 8-bit color depth. At that time it was still the only way to get eight characters, upwards of 30 monsters, and upwards of 100 missiles all interacting on the screen at one time without sacrificing detail and atmosphere.

The graphics criticism caught us by surprise. We thought (and still think) that the game looked great. We probably should have built in a scaling technology to take advantage of hardware that could display the same graphics at higher resolutions. In any case, DIABLO II will probably be our last 2D game.

4. Tools. We developed the original DIABLO with almost no proprietary tools at all. We cut out all the background tiles by hand and used commercial software to process the character art. Spells and monsters were balanced by verbal estimates ("Hey, lets make the lightning about ten percent weaker."). DIABLO II's vastly increased scale required much better tools, and we made some, but not enough.

In many cases we created tools to speed up content creation, but then abandoned them. Too often, we decided to keep using inferior tools because we thought we were almost done with the game, and didn't want to take the time to improve them. Unfortunately, in most of these cases, we were not really almost done with the game, and in retrospect a couple of weeks' worth of work would have helped in the year or more of development remaining.

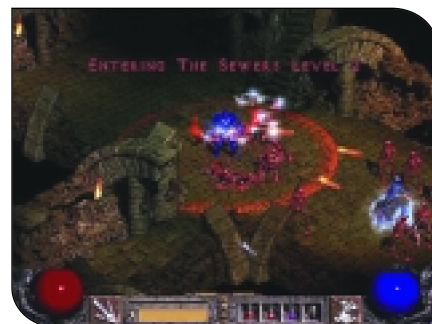
The greatest deficiency of our tools was that they did not operate within our game engine. We could not preview how monsters would look in the environments they would inhabit. We couldn't even watch them move around until a programmer took the time to implement an AI. Even after that, an artist would have to hassle someone to get a current working build of the game to see his creation in action. Our sound effects engineers ended up painstakingly creating .AVI movie versions of animations in order to synch sounds with actions. Our lack of tools created long turnaround times, where artists would end up having to re-animate monsters or make missing background tiles months after the initial work was completed.

We should have made tools that let us create content within the game engine. Instead of just handing off a set of animations and hoping they looked all right when dropped into the game, artists should have had the ability to position and orchestrate their creations themselves. The extra tool development time would have been more than offset by increased efficiency and higher-quality work.

5. Save-game methodology. As much as we tried to make a frustration-free game, we seem to have failed some people with our save-game scheme. Eschewing the common save-game feature we used in the original DIABLO's single-player mode, where every facet of the game state can be saved to files and reloaded at will, we opted to make all modes behave more like DIABLO's multiplayer game. In DIABLO II, we do not save the world state. Reloading the game resets the location of monsters and treasures every time. The character is placed in the town he or she last visited, not in the wilderness or a dungeon.

Although this choice was slightly controversial around the office, it had a lot of advantages. For one, players could not get stuck, unable to progress further. At any time you can restart in town, refight the same monsters for more experience and loot, and return to a difficult area when ready. We created a waypoint system that allows characters in new games to return quickly somewhere close to where they quit the previous game. Finding new waypoints is a rewarding mini-goal during play. We also wanted to discourage the type of play where players feel they must always save the game right before a difficult section, then constantly die and reload until they get lucky and make it through. Finally, it was just easier to make single-player games and multiplayer games work the same way, and multiplayer requires the method we used.

A lot of players don't like our decision. They feel it is too inconvenient to have to fight their way back though the same areas and monsters. Many also want the opportunity to experiment with skill choices and equipment purchases, then later revert the game back to an earlier state if they don't like the results. There are good points on both sides, and we probably didn't spend enough time developing alternatives.



Some of the many locales in Act II: The Sand-Maggot lair (left), Jerhyn's Palace (center), and the Sewers (right). Background elements were created and rendered in 3D Studio Max. The rendered files were cut into tiles and assembled into modular "rooms" with an in-house tile-editing tool. The game engine reassembles the rooms to provide a randomized game environment.

Monsters have 14 possible classifications of animation, from basics such as Walk, Attack 1, and Death, to the seldom-used Block, Run, and four Special modes, reserved for miscellaneous animations. Diablo is the only monster who uses every animation category available.

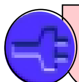
The Final Word

Many more things "went right" than could fit in that section. Our internally controversial plan to tell a separate but parallel story through our cinematic sequences seems to have succeeded, and the workmanship and quality of these sequences has set a new standard. Our marketing and PR departments did a fantastic job building customer awareness and creating a frenzy of interest. DIABLO II's music is outstanding, and along with an amazing array of sound effects, contributes hugely to the atmosphere of the game.

The development of DIABLO II is a remarkable success story. We got the opportunity to make the game we wanted to make — and the game we wanted to play. DIABLO II turned out to be a great game, one that many of us still play every day. Initial sales figures are phenomenal, and reviews have tended to be better than those of its predecessor. We have gained a lot of experience that should help us make even better games in the future.

The only major downside to DIABLO II's development was the inhuman amount of work it required. A yearlong crunch period puts a huge burden on people's relationships and quality of life. Our biggest challenge for the future is figuring out how to keep making giant games like DIABLO II without burning out. As a start, we are hoping our experience will help us do a better job scheduling and managing the workload. We also believe that taking the time to make better tools will make things easier at the end of projects.

Although I tried to avoid personalizing this article, I am extraordinarily proud of the entire development team. DIABLO II could not have happened without all the superb individual efforts, the incredible creativity, and the whole team's dedication to the project, for which they have earned my gratitude, and no doubt that of the legions of players who enjoy the game.



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

Staking Our Own Claim In Entertainment

Odds are, the gaming industry will once again make more money this year than Hollywood takes in at the box office. Yet we still continue to act like we're Hollywood's unpopular, ugly little brother, both by borrowing franchises and brands as well as by mooching Hollywood celebrities to endorse our products. This problem is prevalent at every corner of our industry, from trade shows to journalism to game music.

Consumer gaming magazines are subject to this infection as well. Recently there has been a rash of magazines that claim to not only cover games but also "lifestyle." Lifestyle, to these magazines, apparently means interviews with no-name actors who have nothing to do with gaming whatsoever, and the fact that some of these magazines have folded in recent months suggests consumers may want something else to define the games they play.

At E3, it's a very common sight to see aging, second-rate actors endorsing products, and gaming web sites and magazines eat it up with "celebrity sightings!" Hey, kids, sneak into E3 and you might meet Daisy Duke! Big deal. Are games so unexciting that we need pinup pictures and washed-up actors to draw attention to them?

As an industry, we're afraid to make and market our own brands. A team of talented developers who create their own franchise and want to build a new, original universe will often fight an uphill battle with a publisher who wants to slap an existing brand on it in an attempt to add value.

illustration by Craig Sosonko

So why does our apparent inferiority complex persist? One reason could be that in a game, the real "stars" are the games' characters, not the developers. Some see Lara Croft as the original gaming starlet, which (depending on whom you ask) might be a step in the right direction. Developers are more akin to cinematographers, screenwriters, and directors, who frequently exist behind the scenes and rarely get the credit that a film's stars get. Still, by comparison, these jobs are viewed as more thrilling than game development. A guy sitting at a keyboard in the dark in front of a monitor who works on code is hardly as exciting to most people as a screenwriter or director.

So what can be done about our industry's overall sense of insecurity and inferiority? First, we need to start by making our own franchises. At the same time, we must understand where publishers are coming from and have patience with them. It is your job to convince the publisher that your original brand is better than the existing brand that they want to stick on your title. From their point of view they can have either "existing, proven brand X" or "what's behind door number 2." You'd need

continued on page 63

continued from page 64

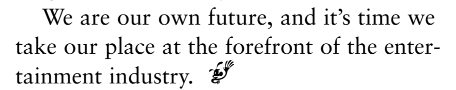
some convincing if you were in their shoes, too. By making our own franchises we'll ultimately make more money instead of paying a steep price for an existing brand that may or may not work. We'll also make better games. Great games stand on their own and sell regardless of existing brands.

Second, we need to do a better job of branding *ourselves* as people and teams. id Software did a great job of building notoriety not only because of the phenomenon that was DOOM, but because of the interesting, dynamic personalities behind it. The flamboyant Romero, the genius Carmack, and the enigmatic Cloud all contributed to and helped define the id brand. Fascinating, real people behind the games help build the brand behind the software.

Finally, we need to continue to raise our standards and production values. Most consumers are better than you might think at judging good production design versus hacked-together scenes, and distinguishing talented voice acting from what your animator's sister can provide. If we meet or better yet exceed their expectations for quality entertainment, then our industry will grow at an even faster rate.

Like most other folks young and old, I enjoy movies. I'll admit that I love going to the theater on a summer night to watch a mindless popcorn flick. However, when I was growing up I wanted to meet Shigeru Miyamoto, not Robert DeNiro. I wanted to meet Richard Garriott, not Steven Spielberg. We need to continue to push our own projects and identities rather than licensing "brand X" or we will become just another platform for the Hollywood advertising machine rather than the genuine art form that is developing and beginning to flourish today.

We are our own future, and it's time we take our place at the forefront of the entertainment industry.



CLIFF BLESZINSKI | *Cliff is 25 years old and already considered an industry veteran. He has been in the industry professionally since the age of 17, when he designed the hit title JAZZ JACKRABBIT for Epic Games. A JAZZ sequel followed soon after, and he then went on to co-design the hugely successful UNREAL and UNREAL TOURNAMENT. Cliff is now lead designer on Epic's upcoming projects.*