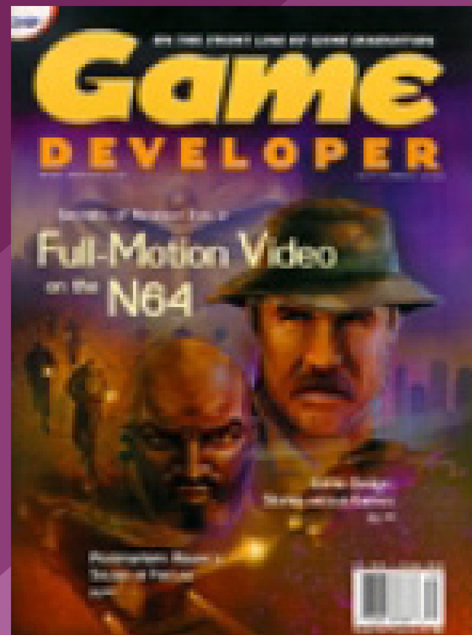
GAME DEVELOPER MAGAZINE

SEPTEMBER 2000



UBM
Tech

Patents, Patterns And Other Patter

I never take joy in devoting column inches in this magazine to fixing mistakes or clarifying issues printed in a previous issue, but it's an unfortunate fact of life. This month it's particularly painful for me, because as we were going to print with our last issue, Brian Sharp discovered that a portion of his article ("Moving Fluid") treads near a software patent held by General Electric. (Regular readers know of my fondness for software patents.) We're grateful to Brian for discovering the similarity and telling us about it, and we apologize that we weren't able to get this information into last month's issue.

The situation is this: the marching cubes algorithm that Brian presented in his article is similar to G.E.'s patent titled "System and method for the display of surface structures contained within the interior region of a solid body" (U.S. patent #4,710,876). It's not clear whether G.E. would pursue a game developer who used the marching cubes algorithm, but we at the magazine felt that we should alert you lest you adapt this algorithm for your game without knowing the extent of the patent. For more details about it, go to www.patents.ibm.com/details?&pn=US04710876__.

While I'm on the subject of software patents, editor-at-large Chris Hecker just alerted me to another patent that we should watch out for. In May, John Nagle of Menlo Park, Calif., was issued a patent titled "Method and system for generating realistic collisions in graphical simulations" (U.S. patent #6,067,096). This one is scary in that (a) it has broad-reaching consequences for most game developers, and (b) Mr. Nagle indicated that he will be pursuing game development companies who violate it. More about this patent can be found at www.patents.ibm.com/details?&pn=US06067096__.

New Patterns Column. This month marks the beginning of a new column for the magazine, written by Chris Hecker and Zack Simpson. Each month, "Game Programming Patterns" will present and discuss an important pattern relevant to game programmers. What, you may be asking, is a pattern? As the column explains this month, a pattern is a named pairing of a problem and its solution. By exploring problem and solution sets within the game programming domain, we hope to help standardize some of the terminology while presenting solutions to common game programming problems. The patterns that Chris and Zack publish, and many more, will be indexed and available on our sister site, Gamasutra.com (www.gamasutra.com/patterns), and we invite you to submit your own patterns to us via a form at that URL. If your pattern is chosen for publication in the magazine, you'll get $100 and the recognition you deserve in print. Check out the new column on page 12.

Join the Discussion on Gamasutra.com. Speaking of Gamasutra, we've had many requests to integrate our magazine content more tightly with our online counterpart. As a result, over the past year we've republished many articles from *Game Developer* on the site, and we take another step this month. At the end of articles throughout this issue, you'll see a little box pointing you to www.gamasutra.com/discuss/gdmag. This is our new online discussion forum, which will hook you up with the magazine's editors, writers, and most importantly, other readers like yourself. We invite you to come to the site to discuss our articles, ask questions, and provide your own insights.

Welcome, Maarten! Finally, to wrap up this month's exercise in three-dot journalism, let me welcome aboard our new Artist's View columnist, Maarten Kraaijvanger. Maarten's day job is at Nihilistic Software, where he serves as lead artist. Now that Nihilistic has shipped VAMPIRE, Maarten's going to share some of the artistic techniques he used during the development of that game, and others to come.

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: 415.905.2228 w: www.gdmag.com

Publisher
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL

Editorial Director
Alex Dunne *adunne@sirius.com*
Senior Editor
Jennifer Olsen *jolsen@sirius.com*
News & Reviews Editor
Daniel Huebner *dan@gamasutra.com*
Art Director
Laura Pool *lpool@cmp.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
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 Microsoft

ADVERTISING SALES

Director of Sales & Marketing
Greg Kerwin e: *gkerwin@cmp.com* t: 415.905.2615
National Sales Manager
Jennifer Orvik e: *jorvik@cmp.com* t: 415.905.2156
Account Executive, Western Region, Silicon Valley & Asia
Mike Colligan e: *mcolligan@cmp.com* t: 415.356.3486
Account Executive, Northern California
Susan Kirby e: *skirby@cmp.com* t: 415.356.3406
Account Executive, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 415.905.2323
Sales Representative, Recruitment
Morgan Browning e: *mbrowning@cmp.com* t: 415.905.2788

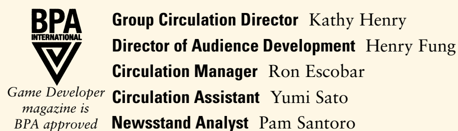
ADVERTISING PRODUCTION

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

CMP GAME MEDIA GROUP MARKETING

Marketing 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

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

SAYS YOU

YOU RANT, WE LISTEN

Keep Cameras Out Of Players' Hands

Jeff Lander's Graphic Content column in the April 2000 issue ("Lights... Camera... Let's Have Some Action Already!") was a good introduction to Hollywood camera techniques and I definitely agree that better camera usage could add more emotion to games. However, it seemed like he was skirting the major issue with camera AI: interactivity. The primary reason that most games don't do dramatic camera cuts at emotional moments isn't ignorance of Hollywood techniques or lack of AI, it's the utter confusion and disorientation that any camera cut can cause when the player is controlling the movement of a character. Consistent game controls require a relatively consistent camera angle.

Look at one of the classic 3D movement games: SUPER MARIO 64. The most annoying part of the whole game is when the camera decides to move around while the player is trying to walk across a narrow path. "Up" suddenly becomes "right," and the little Italian plumber falls down the hole. Everybody hates it when little Italian plumbers fall down holes, but this is the absolute worst way to die in the game; not due to any deficiency in skills or timing, but simply because the game decided to shift your viewpoint (and thus controls) without warning. Direct cuts would be worse.

So overall, good article, but there are only a handful of games that can use these camera techniques during gameplay without hopelessly confusing and annoying the player. If someone finds a way around this, great, but until then I'm going to stick with my unemotional (but controllable) fixed camera angles.

Tom Smith
High Voltage Software
via e-mail

AUTHOR JEFF LANDER RESPONDS: *I agree with you that loss of camera control can be confusing for the player. We had a real big problem with that in a PSX game I worked on. It was particularly a problem when the camera crossed "the line" I talked about in the article. It was so bad at times we were forced to put a big arrow over the character so the player could find him on the screen. It was a design flaw in some of the game levels that was really impossible to fix in post. However, when done correctly, a well-placed camera can really help the player. I saw* RAYMAN 2 *at the Game Developers Conference this past March and its developers had done an effective job of selecting good camera controls.*

That said, I believe that there are ways to add cinematic elements to game camera models. In my current project, I'm looking at interactive cinematography with those exact problems you mentioned in mind. One approach that we are getting some good results from is using a traditional tethered camera while the player navigates the environment. When the player stops to look around, however, the camera "drifts" to preset camera positions in the local area which the choreographer has chosen to best show the environment. When the player encounters another character, the dialogue engine engages. This shows the dialogue in the dramatic manner I discussed in the article. If at any time the player moves the character away from the conversation, this triggers an end sequence to the dialogue like, "See you later." The player then regains a tethered camera. This same kind of thing happens if the player approaches an interesting object. We are calling this system "intention-driven choreography." Of course, I have no idea yet whether players will like it. I do hope to find out soon.

My goal for the article was to get people to think about new ways of approaching camera modeling. Whether it works for your game or not is definitely something you should decide.

More Money Doesn't Equal Better Games

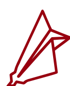
I hate to disagree with you, Mr. Molyneux, but I do not look at the Japanese development houses with awe and dread as I realize the eyebrows on our main characters don't comprise individual hairs ("Next-Generation Gaming," Soapbox, June 2000). I understand what you're saying, I just don't agree with it. There is absolutely no doubt that gaming and other media such as film are merging closer and closer, but is this a good thing? In filmmaking the first rule of thumb is, "Effects are used to accent, not to be the core of the movie." If companies stopped putting so much emphasis on flashy add-ons like FMV and other disproportionately expensive areas of game development that contribute so little to the game, costs could be severely lowered.

As developers learned a few years ago, more than a few gamers couldn't care less about FMV. If it's in there, great, they'll look at it. If not, no one seems to miss it very much. Some big companies use insane amounts of FMV to cover up the fact that their games lack content. They spend so much time and money on effects that their gameplay suffers. I'm not trying to take pot shots at anyone, but clearly there are companies out there that have lost their focus on creating games. They're quite happy simply milking their golden cows.

But where do you come up with this $4–10 million figure to produce a game? My company is a very small and streamlined development house. We have a rock-solid team and we all work very hard and very well together. We're not a public company and we have no VCs funding us. We developed our own tools and our main engine is easily customizable to handle the onslaught of technology. For our largest project, we estimated a year and a half for development at a total development cost of $600,000. Our other two projects can be produced in under a year with a tag of $350,000 or below. People who think that making a good game mandates a $4–10 million budget need to take a good look at their company and consider streamlining.

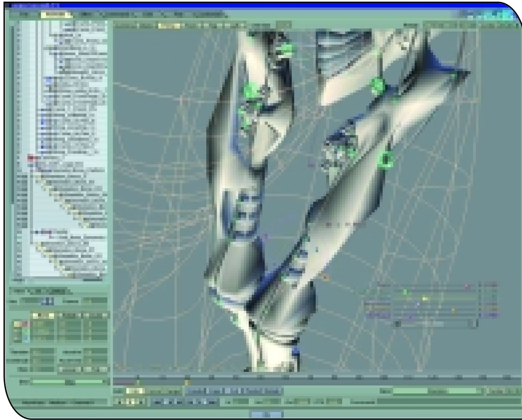
If the game development community is going to continue following the movie industry format, bloated budgets and all, we ought to keep in mind two movies: *Waterworld* and *The Blair Witch Project*.

Jag Jaeger
JV Games
via e-mail



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

PROJECT: MESSIAH CD RELEASE



The creators of Project: Messiah have announced their first full CD release of their Lightwave animation plug-in. Version 1.5.5 adds Lightwave 6 compatibility as well a new effect called Motion Blender. Motion Blender allows animators to point and click to assign numerous poses, including facial expressions and phonemes, to on-screen sliders which are then moved or controlled procedurally to animate. Another new feature is "Use Computed Values," which allow users to create a keyframe based on the position of an object that has expressions or effects acting upon it. Also new for version 1.5.5 is the ability to save OpenGL previews as .AVI files, enabling users to see previews in real time even on slower graphics cards, as well as allowing them to save different versions of motion tests without having to render them. Project: Messiah 1.5.5 has a retail price of $695 but is a free upgrade for current users.

Project: Messiah 1.5.5 | Project: Messiah | www.projectmessiah.com

NEW WINDOWS CE

Microsoft has released Windows CE 3.0, its operating system for appliances and portable devices, and has slashed the price of real-time licenses by up to 50 percent. The new version of Windows CE has been enhanced to provide better real-time capabilities, improved multimedia functionality, and additional language support. In trying to develop a more flexible system, Microsoft has also increased "componentization," which means the operating system can be broken up into pieces and used by developers for specific needs. Windows CE 3.0 also addresses instability concerns and the lack of true real-time capabilities that slowed adoption of previous versions. While Microsoft did not disclose licensing fees, it said it would be offering a free upgrade for users of its Platform Builder 2.12 development tools.

Windows CE 3.0 | Microsoft
www.microsoft.com

WAP BITMAPS FROM MORPHEME

DWBMP, a WAP (wireless application protocol) graphics system from Morpheme, is a Java-based system for rendering dynamic wireless bitmaps (WBMPs). Originally designed to form part of its tool architecture for generating WAP and wireless-based games, Morpheme has decided to make the tool generally available. DWBMP contains a textual creation module that dynamically renders text strings using a series of custom-designed fonts and can be used to add a more interesting look to WAP pages, which are currently dominated by plain text content. The commercial, licensed version of the library offers more features, such as customization of fonts, offline image creation, cropping, clipping, and basic image compositing (displaying text over another image). DWBMP, written entirely in Java, requires version 1.1 or greater of Sun's Java Runtime Environment. Rendering dynamic content online also requires a web server and servlet engine implementing version 2.0 or greater of the Java Servlet API.

DWBMP | Morpheme | www.morpheme.co.uk

GEFORCE COMING TO MACINTOSH

Nvidia is creating a Macintosh counterpart of its GeForce 2 GTS, the GeForce 2 MX, which is designed to bring some of the high-end features and functionality of Nvidia's flagship products to the Mac, and is the first in Nvidia's line to provide native support for Mac color computation. The unit features support for hardware transform, clipping, and lighting, and also supports Nvidia's Natural Shading Rasterizer, which provides enhanced performance for the display of realistic surface textures, shadowing, and lighting. The new chipset will be available both in AGP- and PCI-based configurations.

GeForce 2 MX for Macintosh | Nvidia | www.nvidia.com

CODE PLAY LAUNCHES VECTOR C

Code Play's new Vector C is a compiler for Windows that, as its name suggests, boasts the ability to speed up code by automatically vectorizing standard C code to take advantage of individual microprocessor features. Currently, Vector C supports 3D MMX, 3D Now!, and SIMD extensions, and also integrates into Microsoft Visual Studio. Support for C++ is also planned for the future. Code Play is offering a full-featured Professional Edition for $750 and slightly limited student Special Edition for $80. Both versions are available for download on Code Play's web site.

Code Play | Vector C | www.codeplay.com

INDUSTRY WATCH

THE BUZZ ABOUT THE GAME BIZ | *daniel huebner*

Microsoft Buys Bungie. Rumors have been flying since the Xbox announcement last March that Microsoft was looking to make acquisitions in the game industry. While Square and Eidos were the names being passed around at first, Microsoft's first buy is Bungie. Bungie will become an independent studio within the Microsoft Games Division, though the company's Chicago offices will be relocated to Redmond, Wash. Microsoft will get Bungie's HALO, currently under development, but the all rights and technologies for ONI and the MYTH series will go to former Bungie distributor Take-Two Interactive as part of a deal enabling Microsoft to obtain Take-Two's 19.9 percent stake in Bungie. Though Bungie will continue to make its own decisions in regard to development platforms, including whether or not to continue the company's longtime Macintosh support, the company is expected to play a key role in developing and defining the Xbox platform.

Bungie's HALO: Will it be for Xbox only?

The Bungie buy isn't Microsoft's only deal with Take-Two. Microsoft is choosing Take-Two subsidiary Broadband Studios to provide online distribution of selected Microsoft game titles. Under the deal, Microsoft titles will be broadcast over Broadband Studios' Power Play network, and the two companies are also looking into creating an online game channel built around Microsoft properties.

Disney Broadband. Disney is also entering the broadband fray. Disney Interactive and Into Networks are teaming to create the Disney Interactive Channel on Into's Play Now network. The channel will deliver entertainment products from Disney Interactive over Into's broadband content platform, enabling users to try titles before buying, rent titles, or pay for monthly channel subscriptions. Disney Interactive will launch its broadband channel with a selection of titles, including its A BUG'S LIFE ACTION GAME.

Other Acquisitions. The spirit of acquisition and consolidation is burning brightly in the game industry. Despite persistent reports that they are in the middle of an effort to purchase struggling Eidos in a $1 billion deal, Infogrames found enough spare change to snap up Texas-based Nintendo 64 development studio Paradigm Entertainment in an effort to shore up Infogrames' next-generation console offerings.

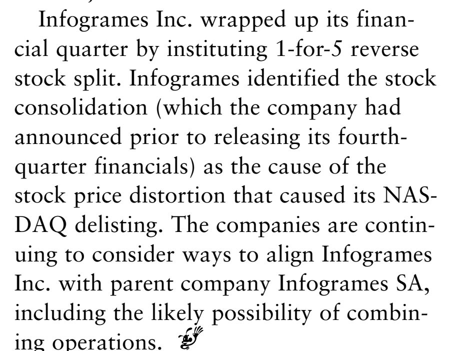
Tool companies are also consolidating in order to expand their product lines. Dublin-based physics SDK maker Havok is buying out German rival Ipion, intending to merge the strengths of the two existing physics simulation kits into a single segment-leading product. A similar strategy is behind Avid's purchase of The Motion Factory. The goal of the purchase is to unite the interactive 3D animation technologies of The Motion Factory with Avid subsidiary Softimage's top-of-the-line XSI tools. Former Motion Factory president and CEO David Pritchard will oversee the integration from the post of general manager of Softimage.

THQ Rights Plan. While some companies are buying, others are looking to avoid being bought. THQ is following Activision and Acclaim in adopting a shareholder protection plan. Like the other plans, THQ's shareholder rights initiative is designed to ward off any unsolicited takeover attempts by triggering a shareholder stock payout if any outsider purchases 15 percent or more of the company's outstanding shares. The plan, which the company says is not in response to any specific buyout effort, went into effect July 3.

We All Live in a Pokemon World. Nintendo reports that Pokemon sales continue to soar. Sales of Nintendo's Pokemon videogames for the first five months of the year are up 220 percent from the same period last year. For the year, Nintendo expects total revenues to top $3 billion in North America alone. Besides the new animated feature film that hit theaters in July, Nintendo plans to fuel the flames by releasing four new Pokemon games before year's end, plus hundreds of licensed products.

GT Becomes Infogrames Inc. While Infogrames SA continues its acquisitive ways, last year's buyout of GT Interactive has yet to bear fruit. GT Interactive, now known as Infogrames Inc., is reporting a sharp decline in fourth-quarter revenues for the period ending June 30. The company brought in $61.1 million for the quarter, compared to $93.5 million in the same period last year. The net loss for the quarter, due in part to charges for restructuring and reorganizing (including layoffs and the closure of GT's international offices), stretched to $140.6 million from last year's $54.8 million. Excluding charges, however, Infogrames Inc.'s fourth-quarter loss totaled just $6.6 million.

Infogrames Inc. wrapped up its financial quarter by instituting 1-for-5 reverse stock split. Infogrames identified the stock consolidation (which the company had announced prior to releasing its fourth-quarter financials) as the cause of the stock price distortion that caused its NASDAQ delisting. The companies are continuing to consider ways to align Infogrames Inc. with parent company Infogrames SA, including the likely possibility of combining operations.



DIGITAL VIDEO EXPO & WEB VIDEO EXPO
LONG BEACH CONVENTION CENTER
Long Beach, Calif.
October 2–6, 2000
Cost: $40–$1,495
www.dvexpo.com

PROJECT BAR-B-Q
GUADALUPE RIVER RANCH
Bourne, Tex.
October 19–22, 2000
Cost: $2,100
www.fatman.com/bbq.htm

Game Programming Patterns & Idioms

Every game programmer has his or her bag of tricks and techniques. Almost every programmer agrees that sharing these combinations of problems and solutions is an important part of learning and growing professionally. Unfortunately, the lack of a common vocabulary makes sharing information much less efficient and more difficult than it needs to be. An "overlay" at one company is a "gump" at another. ("Gump" is the actual name of a windowing system at a game company that shall remain nameless. We ignore the fact that one of your humble editors is at fault for the name.) Likewise, one developer's "clipping" may be another's "collision detection."

This lack of vocabulary is a sign of our youth as an industry and a profession. Do you think that civil engineers bicker over the meaning of an aqueduct or an overpass? How much unnecessary mental energy do we spend arguing with our peers over what turns out to be a simple difference in definitions?

Vocabulary is more than mere communication — it is knowledge. When something has an accepted name, it also tends to have associated theory and lore. If you know the name of something, you will hear rumors about it, you can search the Internet for it, and you can read books about it.

How can we as game programmers develop a vocabulary to accelerate the process of sharing information and communicating about problems and solutions, making our games better and creating them more efficiently in the process? This column aims to help, with you, the reader, playing an important part. Every month we'll print a "pattern" that captures and discusses an important problem in game programming. These patterns will be submitted by game developers (again, that means you). They will be catalogued in a database on Gamasutra.com, where developers can search for patterns solving their problem, and add and annotate the patterns based on their experience. If this works, it'll be a great resource to help both new and experienced game programmers ply their craft.

Patterns?

Patterns have gotten a lot of hype lately, but underlying the hype is the fundamental concept of a vocabulary. A pattern is nothing more than a problem and solution pair which have been given a name we can use to discuss it. For example: "An overpass is a structure which allows cars to pass over one another thus avoiding time wasted at stop lights or stop signs." A good pattern avoids questions of implementation. It doesn't say, "An overpass must be built of prestressed concrete beams, last 150 years, and be weatherproof." It might talk about common problems: "Overpasses tend to take up a lot of space and be costly." A pattern is just a formalized description of a problem that has been frequently solved but under different names.

There are numerous existing software pattern collections currently available and much has been written about patterns in software and other industries. (See For More Information for links.) Some of these collections are extremely formal, sometimes to the point where the formalism gets in the way of the underlying idea. We feel formalizing patterns to the point where they start to read like a programming language is a mistake. In this column, we plan to keep things casual and, we hope, intuitive.

However, the most important characteristic of this column is that the patterns will be submitted by readers. The entire point of the column and database is to share information about common idioms and techniques among all developers, and that means all developers have to participate and contribute the patterns. Take a look at our format on the facing page, and submit your patterns in this format to the e-mail address provided. Keeping the length down will help us fit the pattern on a single page and help you focus on what's important. We'll publish one pattern a month in the column and even more to the searchable database, and we'll begin developing the vocabulary we need to advance the state of our art at a much faster pace.

One final note: don't be surprised if your first reaction to some of the patterns is, "Duh, everybody knows that one." That's O.K., since we're trying to develop a shared vocabulary for discussing common game programming problems. Not every pattern will be new to you, but hopefully every pattern will encourage you to think about its problem. Maybe you know another name for the pattern that we can add to the database. Maybe you have a better example. Or, maybe know an even better solution. Submit your knowledge and we'll all learn!

FOR MORE INFORMATION

The Patterns Home Page
http://hillside.net/patterns

Zack's Patterns Pages
www.totempole.net/gamepatterns.html
www.totempole.net/gameprocess.html

CHRIS HECKER | *Chris Hecker (checker@d6.com) is editor-at-large of* Game Developer.
ZACHARY BOOTH SIMPSON | *Zack is the former director of technology at Origin and Titanic. He currently wanders the countryside crushing cars and terrorizing suburban dwellers.*

Spatial Index

a.k.a. Spatial Partition, Spatial Datastructure, Geometric Datastructure

Problem

Almost all games have a database of world objects, even if it is as simple as a static array or a linked list. Sometimes the world itself is part of this database. Queries into the database are often based on a spatial key, like a position or a region in the world. Often these queries must be very fast because they are made many thousands of times per second. Examples: finding all the objects near an explosion and deducting hit points; finding all surfaces visible in a view frustum for rendering; finding the item that intersects the last mouse click. One correct but slow way to implement spatial queries is to iterate through every object in the database, checking it against the spatial key.

Solution

A Spatial Index accelerates spatial queries. Given a spatial key, the Spatial Index allows quick access to the database's associated values. The Spatial Index may cache information from the database to speed queries. It may refer directly into the database's values, or it may return a key which can be used to attain the value from the database. There are many implementations of a Spatial Index, but most spatially subdivide the world to reduce the search space, sort the objects spatially, or use time- or space-coherency between lookups to speed spatial queries — or some combination of all these techniques.

Examples

The Spatial Index is among the most studied and discussed patterns in game development. A very abbreviated list of implementations includes: binary space partition trees, room/portal systems, X/Y hash with chained collision resolution, and octrees.

Issues

Conceptual confusion. A database index is not a database and vice versa. A single database may have multiple indices or no indices at all. An index is conceptually distinct from the database itself, and is an additional datastructure used to speed up queries on the database. For example, a book may have a keyword index, a table of contents, and an index of figures, or it may have none of these. In each of those examples, the text of the book is the database. We use a different index depending on what type of key you want to use to look up a page in the book. For the table of contents, the key is a concept. For the keyword index, it's a word, and for the figure index, well, it's a figure. You get the idea. A game may have an object database with an associated Spatial Index for the renderer, a different Spatial Index for the collision detector, and a "type index" so AIs can query based on the different types of bad guys.

Coupling. Spatial Indices have a tendency to become too tightly coupled with their associated database, which can reduce flexibility and reusability. The index can be overly optimized for a particular subsystem leaving it less useful for another subsystem. For example, if the Spatial Index is highly optimized for and coupled to the renderer, the AI subsystem might have trouble using it to find the nearest enemy units efficiently.

Synchronization and duplication. The keys in an index must remain synchronized with the values in the database. This can be difficult if the index is cached to improve performance. If a unit moves in the database, the cached data in the Spatial Index must either be updated or invalidated. Uncoupling an index from a database can create synchronization problems. Data is often duplicated from the database to the index, which consumes resources (both memory and overhead keeping the data synchronized).

Related Patterns

In the future, this section will point to other related patterns in the database, but since this is the first one, there's nothing here! See the box below on how you can contribute to help out this poor and lonely section.

Uses, Credits, and References

This pattern is widely used throughout the industry. See *The Design and Analysis of Spatial Data Structures* by Hanan Samet (Addison-Wesley, 1990). Also, Mike Abrash describes QUAKE's Spatial Index, called the Potentially Visible Set, on Blue's News at www.bluesnews.com/abrash. The PVS is actually a Spatial Index that references a BSP, another Spatial Index.

Now It's Up To You!

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

Physics Engines
Part One: The Stress Tests

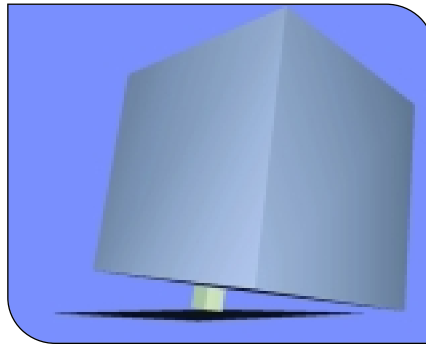
by jeff lander & chris hecker

As devoted readers of *Game Developer*, you are all aware of our belief (or hope!) that realistic physics can greatly improve the gameplay in interactive entertainment. You also know that creating a robust and flexible physical simulator is difficult work. Simulators are hard to design, challenging to code, and even more difficult to debug. These challenges cause many programmers and producers to look for external help for their physics development problems.

Lately we have seen the emergence of licensable "physics engines" from companies wishing to fill this need. We feel the market has matured to the point where it is time for us to take a hard look at these products to see how well they work.

We will be looking at three products that offer roughly the same level of technology and features: MathEngine's Dynamics Toolkit 2.0 and Collision Toolkit 1.0, the Havok GDK from Havok, and Ipion's Virtual Physics SDK. Our focus is on rigid body dynamics. We will not review cloth, particle, water, or other kinds of simulation, even though these packages support some of those features.

This month, in the first of two parts, we have created a series of tests that stress the capabilities of the simulations in difficult-to-solve situations where physical simulations typically break down. These tests give us a general feel for each engine's implementation of important core features such as contact, constraints, and integration, as well as a working knowledge of each API. Next month, we will take an in-depth look at each package with specific attention paid to how these packages will integrate into a large-scale game project.



The Havok GDK undergoes Test 1, in which a large cube is dropped on a small cube.

Ground Rules & Disclaimers

First, it's important to point out that the stress tests are not exhaustive, nor directly representative of in-game situations. They are simulations of physical configurations specifically chosen to stress the engines in difficult numerical situations.

Our goal with these tests was to break the simulators. We are not necessarily expecting perfection (although it would be nice!). We chose this approach because users and artists have an annoying tendency to create physical configurations that break physics simulators, so it's better to find weaknesses before licensing than during final play-test.

Though the tests are difficult, they are not unrealistically complex. Artists could create configurations that correspond to these tests without knowing the configurations might be problematic. We did not tune the tests to exacerbate a discovered bug. Rather, we made a good-faith programming effort to help each engine simulate the tests successfully, tuning parameters and requesting technical support in some cases.

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.*

When creating the score for each test, we evaluated a specific set of factors:

- How physically plausible and consistent with our expectations were the results?
- How stable were the results? Did the simulation come to rest in a stable state or did the system continually jitter or explode?
- Were the results accurate given the design of the situation?
- How easy were the scenarios to design and adjust in each SDK?

The grading for each test ranges between 0.0 and 1.0, with 1.0 being perfect in all areas. The grades for each engine are absolute compared to a theoretical ideal, not relative to the other engines. A score of 0.5 is the minimum "acceptable" score given our expectations.

We designed the scenarios to be the same scale and with the same units (where possible) in each package. Scale is very important in a simulation, so we used standard units for measurement and chose a world scale that worked well in all three systems.

The final scores are not presented in an easy-to-compare table. This is a conscious decision, as we believe simple numbers are meaningless without a full understanding of the context. Choosing a physical simulator is not as simple as setting up a race between systems and see who wins. There are many other tests that might be more representative of your problem domain, and we encourage you to run them yourself before deciding.

The Tests

Test 1: A large cube is dropped on a small cube. This challenge was designed to test the collision detection and response of each system. Masses and sizes that vary by an order of magnitude can cause trouble for contact solvers and collision detectors.

The cubes were axially aligned and centered at the origin and translated up along

the Y-axis. Both cubes were initially off the ground and dropped at the same time. The large cube was 10m×10m×10m with a mass of 5,000kg and the small cube was 1m×1m×1m with a mass of 5kg. The coefficient of restitution between all objects and the ground plane was set to 0. We expected the two cubes to drop straight on top of each other without bouncing and come to an immediate rest.

MATHENGINE 0.7 | Even though we tried everything to ensure the coefficients of restitution were 0.0, we could not get rid of the bounce when the boxes hit. In this test, the boxes bounced, then the smaller box shot out from beneath the large box.

IPION 0.75 | The boxes dropped directly on each other without bouncing. However, the top box tipped over then jittered a bit before stabilizing.

HAVOK 0.6 | On contact, the top box slid back and forth in a very unrealistic manner before finally tipping over. Setting the friction even higher didn't help. The objects also went to sleep too fast, freezing as they were visibly moving.

Test 2: A slightly larger cube is dropped on a smaller cube. This was a variation on the first test in that the difference between the cubes' sizes was much smaller. This test was meant to be a "gimme" and should just work.

The cubes were set at the origin and translated up along the Y-axis. Both cubes were initially off the ground and dropped at the same time. The large cube was 5m×5m×5m with a mass of 625kg and the small cube was 4m×4m×4m with a mass of 320kg. The coefficient of restitution between all objects and the ground plane was set to 0. We expected the two cubes to drop straight on top of each other without bouncing and come to an immediate rest.

MATHENGINE 0.7 | Again, the initial bounce between the objects caused the top box to tip over.

IPION 0.9 | The boxes dropped and stayed stacked. The system slept early while still jittering slightly.

HAVOK 0.9 | On contact, there was slight sliding between boxes. They stayed stacked and there was no jitter between them.

Test 3: A large cube is constrained to a small cube and both are dropped. This challenge was designed to test the constraint system as well as the collision detection and response of the systems. Again, orders of magnitude uncover numerical problems in constraint solvers and constrained collision and contact resolvers.

The cubes were set at the origin and translated up along the Y-axis. The two boxes were constrained together with a three-degree-of-freedom spherical constraint positioned exactly midway between the two cubes. Both cubes were initially off the ground and dropped at the same time. The large cube was 10m×10m×10m with a mass of 5,000kg and the small cube is 1m×1m×1m with a mass of 5kg. Again, we set the coefficient of restitution between all objects and the ground plane to 0. We expected the two cubes to drop straight on top of each other without bouncing and come to an immediate rest, with the top box balancing on the constraint. The joint should have stayed rigid, always maintaining the separation between the two boxes.

MATHENGINE 0.4 | Again, the two boxes bounced on the drop. The constraint held the boxes apart. However, the box tipped over and the system continued to jitter without coming to a rest.

IPION 0.5 | The joint collapsed on drop. The top box fell over and there was some jitter before the system stabilized to sleep.

HAVOK 0.4 | The joint collapsed on drop. The top box fell over and there was some jitter before the system stabilized to sleep. Applying a mouse force allowed the constraint to be completely violated, adding unrealistic energy to the system.

Test 4: A slightly larger cube is constrained to a smaller cube and both are dropped. This is a variation on Test 3 in that the difference between the cubes is much smaller. Like Test 2, this was another "gimme" with very similar sizes and masses, making for an easy-to-solve configuration.

The cubes were set at the origin and translated up along the Y-axis. The two boxes were constrained together with a spherical joint positioned exactly midway between the two cubes. Both cubes were initially off the ground and dropped at the same time. The large cube was 5m×5m×5m with a mass of 625kg and the small cube was 4m×4m×4m with a mass of 320kg. The coefficient of restitution between all objects and the ground plane was set to 0. We expected the two cubes to drop straight on top of each other without bouncing and come to an immediate, balanced rest. The constraint joint should have stayed rigid maintaining the separation between the two boxes.

MATHENGINE 0.7 | The boxes bounced, but otherwise perfect.

IPION 0.8 | There was a slight bounce. The top box finally fell over in a plausible manner.

HAVOK 0.75 | There was a slight bounce. The top box finally fell over and the system went to sleep a bit too early. We had to adjust default parameters to get this result.

Test 5: A large cube is constrained to a world anchor. A small cube is constrained to the large cube. This challenge was designed to test the constraint system with objects having a great difference in mass, including the infinite mass of the world constraint.

The cubes were set at the origin and translated up along the Y-axis. The two boxes were constrained together with spherical joint positioned exactly midway between the two cubes. The top large cube was constrained to a world anchor above it by another spherical joint. The large cube was 10m×10m×10m with a mass of 5,000kg and the small cube was 1m×1m×1m with a mass of 5kg. We expected the objects not to move, as the constraints would hold everything exactly in place.

MATHENGINE 0.95 | Behaved as expected.

IPION 0.95 | Behaved as expected. Due to the interface, we couldn't exert a strong mouse force on the small object to test stability.

HAVOK 0.9 | Behaved as expected. However, forces applied to the small box made the system slightly unstable.

Test 6: A small cube is constrained to a world anchor. A large cube is constrained to the small cube. This challenge was the same as the previous test with the two boxes reversed. It was designed to test the constraint system with objects having a great difference in mass, with the small mass between the large and infinite mass.

The cubes were set at the origin and translated up along the Y-axis. The two boxes were constrained together with a spherical joint positioned exactly midway between the two cubes. The top small cube was constrained to a world anchor above it by another spherical joint. The

large cube was 10m×10m×10m with a mass of 5,000kg and the small cube was 1m×1m×1m with a mass of 5kg. We expected the objects not to move, as the constraints would hold everything exactly in place.

MATHENGINE 0.9 | There was a slight energy addition, but it was very solid.

IPION 0.45 | The constraint was violated in a springlike stretching manner. The system didn't blow up.

HAVOK 0.4 | With the default constraint stiffness parameter, tau, the constraint was stretched in a springlike manner and the system even went to sleep with the constraint violated. When we increased the tau, there was only a slight jitter. However, applying any force to the big cube caused the system to gain energy and explode.

Test 7: A small cube and large cube are dropped into a gap between two planes that narrows gradually. This challenge tested the collision detection system and the contact resolution system. The gradual slope of the planes caused the contact and collision system to exert large normal forces to stop the cubes from falling.

Two large and narrow boxes used as planes were positioned so they formed a vertical chute that narrowed to a point at the bottom at a 5.625-degree slope, and the two cubes were dropped into the gap. The large cube was 10m×10m×10m with a mass of 5,000kg and the small cube was 1m×1m×1m with a mass of 5kg. We expected the objects to fall into the gap, become wedged between the planes, and not move.

MATHENGINE 0.8 | There was an initial bounce, but otherwise it performed well. When the small box was moved with a force, energy was occasionally added. The lack of some basic math functions in the SDK made the test difficult to create.

IPION 0.0 | This was a complete failure. Contacts were missed or incorrect impulses were applied and the boxes fell through the chute.

HAVOK 0.9 | Almost perfect. There was a small amount of bounce.

Test 8: One cube is dropped so that it will collide with another cube on collinear edges. This challenge tested the collision detection system. Degenerate collision manifolds can be problematic.

A slightly smaller cube was dropped onto a larger one so that they struck edge to edge. The small cube was 4m×4m×4m with a mass of 320kg and the large cube was 5m×5m×5m with a mass of 625kg. We expected the objects to fall and make contact at the edges without bouncing and then come to an immediate rest. Due to numerical inaccuracies with rotating the cubes to set up the test, we couldn't be totally sure the boxes hit exactly edge to edge.

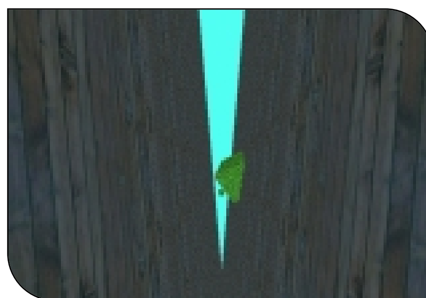
MATHENGINE 0.7 | The dropped box slid off to the side in an unrealistic, slippery manner.

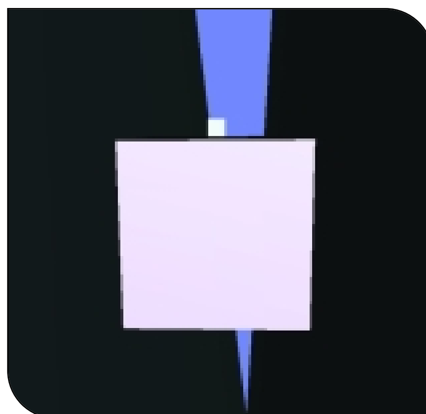
IPION 0.95 | Almost perfect, but it went to sleep in dynamically unstable situation. A small epsilon gap was visible between the two cubes.

HAVOK 0.9 | Almost perfect, however there was a small amount of bounce. An epsilon gap was visible between the objects.

Test 9: One cube is dropped on another cube of the same size, so that they will collide exactly point-to-point. This challenge tested the collision detection system as another degenerate collision manifold test.

Two identical cubes were dropped so that they struck point-to-point. The cubes were each 4m×4m×4m with a mass of 320kg. We expected the objects to fall and contact at the point without bouncing and then come to an immediate rest. As in the previous test, due to numerical inaccuracies with rotating the cubes to set up the test, we couldn't be completely sure the boxes hit exactly point-to-point.





Cubes drop into a chute in Ipion (top) and Havok (bottom) in Test 7.

MATHENGINE 0.7 | The dropped box slid off in an unrealistic manner, exactly like in the previous test.

IPION 0.85 | There was a little bounce and they went to sleep in dynamically unstable situation.

HAVOK 0.9 | Almost perfect. The boxes sat there stably, eventually tipping over. However, the default epsilon gap between objects was visible.

Test 10: Boxes are dropped to form an uneven stack. This challenge tested the collision detection system with massive numbers of contacts and collisions.

A series of boxes of various sizes were dropped onto each other with the initial rotations around the Y-axis, and slightly offset from the center. The goal was to try stacks of up to 40 boxes. We expected the objects to fall into a stable stack of blocks that behaved in a believable manner.

MATHENGINE 0.2 | When more then 10 to 12 boxes were dropped, the system crashed. Otherwise the boxes bounced badly, even adding energy. The system was very unstable, though no boxes appeared to interpenetrate.

IPION 0.6 | Stable, but really started slowing as boxes were added to the system. Some boxes eventually fell through each other and the boxes at the bottom jiggled a bit. Parts of the stack slept while others jittered and interpenetrated.

HAVOK 0.75 | Perfectly stable for a while, but with the maximum stack, the boxes started falling through each other and the floor.

Test 11: A hinge joint with similar bodies. This test should have been easy to solve. A one-degree-of-freedom hinge constrained two similarly sized bodies.

The cubes were set at the origin and translated up along the Y-axis. The two boxes were constrained together with a hinge joint positioned exactly midway between the two cubes. Both cubes were initially off the ground and dropped at the same time. The large cube was 5m×5m×5m with a mass of 625kg and the small cube

was 4m×4m×4m with a mass of 320kg. The coefficient of restitution between all objects and the ground plane was set to 0. We expected the two cubes to drop straight on top of each other without bouncing and come to an immediate, balanced rest. The constraint joint should have stayed rigid, maintaining the separation between the two boxes. We should have been able to move the bodies and the hinge should have only rotated around its axis.

MATHENGINE 0.7 | The boxes bounced, but otherwise ran perfectly. The lack of mouse forces in the demo made it slightly hard to test the hinge.

IPION 0.8 | There was a slight bounce. The top box finally fell over in a plausible manner. It was very stable.

HAVOK 0.7 | There was a nonstraight bounce. The top box finally fell over and system went to sleep a bit too early. Havok doesn't support hinges except through linearly dependent spherical joints.

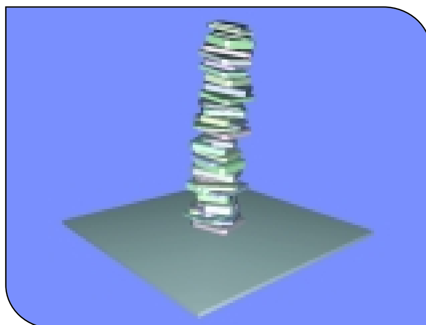
Test 12: A hinge joint with differing masses. This was the "hard" version of the previous test. The order of magnitude differences between the bodies can cause numerical trouble.

The cubes were set at the origin and translated up along the Y-axis. The two boxes were constrained together with a one-degree-of-freedom hinge constraint positioned exactly midway between the two cubes. Both cubes were initially off the ground and dropped at the same time. The large cube was 10m×10m×10m with a mass of 5,000kg and the small cube was 1m×1m×1m with a mass of 5kg. The coefficient of restitution between all objects and the ground plane was set to 0. We expected the two cubes to drop straight on top of each other without bouncing and come to an immediate rest, with the top box balancing on the constraint. The joint should have stayed rigid, always maintaining the separation between the two boxes.

MATHENGINE 0.45 | The two boxes bounced on the drop. The constraint held the boxes apart. However, the box tipped over and the system continued to jitter without coming to a rest.

IPION 0.45 | The joint collapsed on drop. The top box fell over and jittered.

HAVOK 0.4 | The joint collapsed on drop and the top box fell over. Applying a mouse force allowed the constraint to be completely violated and the system could blow up.



MathEngine (top) and Havok (bottom) handle stacking blocks (Test 10) with varying success.

Things We Didn't Test

Space and time limitations prevented us from testing more potential problems. Other things we could have tested include:

- Other joint types; two-degree-of-freedom rotational joints are often problematic, prismatic joints with different moments of inertia, and so on.
- Stiff springs and user-defined forces; jerking the constraints around was not really tested.
- Concave collision detection, union of convex collision detection.
- Sliding bodies over one another looking for inconsistent normals, especially as contact manifold collapses from a 2D patch to a 1D line segment and then to a 0D point.
- Tunneling of fast-moving objects or objects strongly pulled into the ground (preliminary tests in this area indicated problems with some of the engines).

Conclusions

After conducting all of these tests on each system, we were impressed with the general engine stability across the board. With the exception of the box-stacking test on MathEngine, which was likely a memory problem, the simulators were able to handle every test without crashing. We had some difficulties with the access functions in some cases, which we will cover in depth in next month's conclusion. Nevertheless, we were able to get every test running fairly rapidly.

MathEngine is very capable at handling hard constraints. The bouncing problem on collisions hurt its score on most tests. After talking with MathEngine technical support about it, we adjusted the gamma and epsilon values and were able to improve several of the tests. However, this required case-by-case experimentation and never completely eliminated the bounce. The collision detection and resolution worked well; we never saw objects interpenetrating.

The Ipion engine was a solid performer as well, with the exception of Test 7, the narrowing chute. The rigid constraints were not exactly rigid but they were pretty stable. The collision detection and handling were reasonable, excepting the chute and box-stacking tests. The system also slowed down to unacceptable levels as the number of interacting objects increased.

The Havok engine posted generally strong scores. They clearly need to work on their rigid constraint system. The default constraint tau value (which is not well documented) resulted in springlike constraints that didn't perform well in our tests. Increasing tau caused the constraint to become more rigid, however the system then became unstable. We were also able to see object interpenetration problems when the system was really stressed. That said, the collision system was very robust and fast, handling even fairly large block stacks without slowing much.

So, which package is right for you? These stress tests are only the first part of an answer to that question. Next month we will have an in-depth review of each simulator, covering the complete feature sets, documentation, and API issues.

NOTE: As we were running these tests, Havok announced that they had purchased Ipion in a consolidation of this newly formed market. They will continue supporting both packages until they can consolidate them into one system.

"Art and Intelligence": 3D Painting

Creating high-quality 3D graphics and animation requires a great deal of artistic expertise. This expertise comes not only from the artists who will create the pieces that will produce the final picture but also the programmers who create the technology behind the images. Many people don't think of game programmers as artists per se, however, there is definitely an art to the technology. That is not to say that you can necessarily program good art. We have all used the Adobe Photoshop filters that can make a photo look like it was drawn with a crayon or painted with watercolor. Such effects are accomplished with an algorithm that mimics the characteristics of these natural media. While the processed images can resemble the look created by an artist, few would argue that these image manipulation tricks are actually creating art.

Lately, I have been considering something Pablo Picasso said: "There are painters who transform the sun into a yellow spot, but there are others who, with the help of their art and their intelligence, transform a yellow spot into the sun."

The technology behind a Photoshop filter clearly takes the former approach of changing the sun into a nice yellow spot. Many of the "artistic" tools created for use in production adopt this approach. While they are capable of creating nice images, it makes me wonder if they are really creating "art."

In my work, I have always believed that my job as a programmer of 3D technology was to create tools that enabled the creation of new, compelling art. I never want to be in the business of simulating the artistic process, but I want to support and nurture it. I want to enable people to achieve their visions more easily and so I try to develop technology to support this goal.

JEFF LANDER | *When not trying to draw a sun with the broken yellow crayon from his Crayola Growing Kid's Washable 8-Pack, Jeff can be found napping on his towel at Darwin 3D. You can poke him at jeffl@darwin3d.com.*

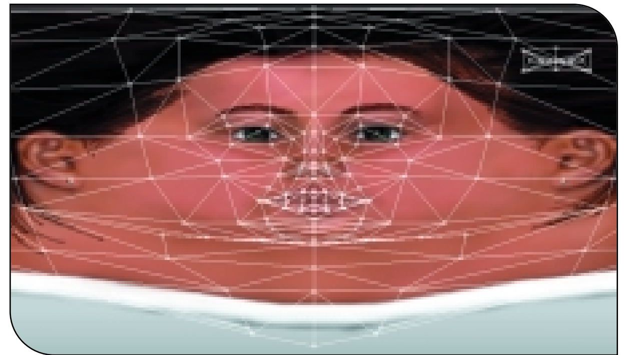


FIGURE 1 (top). A robo-painting of the author and his wife.
FIGURE 2 (bottom). A model unwrapped as a texture guideline.

Painting the Vision

One area where I see quite a large breakdown in the creative process is in the creation of 3D models. Creating a 3D model is largely done as a multi-step process. First, an object or character is designed. This can be a sketch on paper, a sculpture, or an idea in an artist's head. Next, an artist turns the design into a 3D object in a modeling package. To give the object detail, an artist creates a texture map either by painting a texture or manipulating digitized images in a tool such as Photoshop.

The artist then applies these textures to the object by projecting them onto the object algorithmically and painstakingly adjusting the map to make a good fit. The process of painting and mapping is then iterated to make sure everything is in the right place and looks correct. This can obviously take a great deal of time in many cases.

To speed up this process, some companies have tried reversing the last two steps. The texture map is applied before it has been created. Once the texture is applied and then unwrapped with wireframe guidelines, the result is a template that shows the relationship between the texture and the object. You can see an example of this in Figure 2. An artist can then paint the texture within the guidelines, hopefully limiting the number of iterations required.

While often effective, this method seems to me to be very backwards artistically. We have constrained the artist to work with a flattened and skewed vision of what the

final object will look like. It's obviously quite possible to create great-looking 3D objects with this method. Just look at all the great art in many of the games out there. Clearly artists have learned to work within the constraints imposed by technology.

Graphics hardware is now capable of rendering multiple simultaneous textures, as well as special effects such as real-time bump and environment mapping. As this technology becomes a standard part of game development, the process of creating textures to support the hardware will become more important then ever. Obviously it's possible to create all these textures using the traditional production pipelines. However, we should continue to ask if there could be a better way.

These problems have been obvious ones for some time now. In 1990, Pat Hanrahan and Paul Haeberli observed that it would be better if it were possible to paint directly on a 3D model. At Siggraph that year, they described a method for direct WYSIWYG (what you see is what you get) painting and texturing on 3D shapes. Software developers immediately saw this as a potentially important evolution in graphics production and many 3D paint packages were born.

With a 3D paint system, users are able to manipulate objects in 3D much as they would in a modeling program. When the user paints color directly on the object in the 3D view, this paint information is converted internally to a 2D texture map. The more advanced 3D paint packages out there allow you to create multiple layers of textures and paint them with a variety of brushes and stroke styles. It's even possible to paint a bump map directly and see the results change interactively. You can see an example of a character head being painted in Metacreations' Painter 3D (recently sold to Corel) in Figure 3.

You'd assume this technology would go a long way toward speeding up the creation of a 3D model. However, it seems as if these 3D paint packages have yet to prove themselves. I haven't seen them widely used by many artists and I'm not sure why.

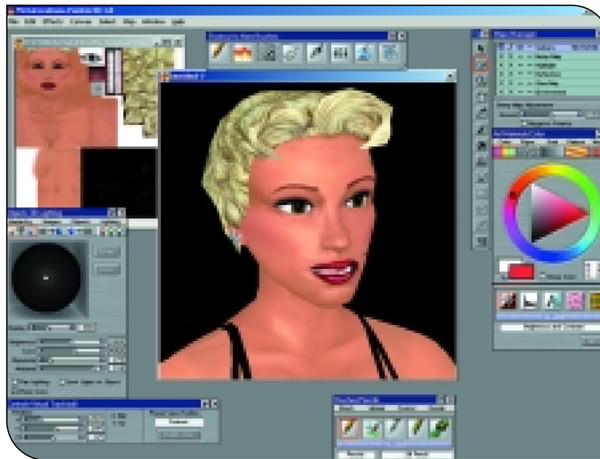


FIGURE 3. A character in Painter 3D.

One reason may be the cost. Good 3D paint packages are not cheap. In game companies it can be difficult for an artist to get a copy of a good animation system and getting the company to shell out an extra $500 or more for a paint system may be asking too much. It's also yet another tool to learn which may or may not integrate well with the existing production process. Many artists may simply not want to bother and stick with whatever has worked for them in the past.

How Does It Work?

The idea of 3D painting is certainly interesting enough to investigate further. It would be very handy, for example, to be able to touch up a texture map from within my model viewer. In the simplest case, I might have a 3D model with a texture already applied and I want to touch it up by adding a mole or wrinkle lines.

In order to paint on the 3D surface I have to know what point on the model is directly under my mouse when I press the button. It was only a few years ago that I was still writing my own software renderer (remember those days?). In this renderer, I had a routine that actually drew the pixels for each polygon. If I moved the mouse across the screen, I could just figure out what was drawn under it since my little software routine drew every pixel. Thus, 3D painting was pretty easy to implement, it just wasn't very fast.

Now we have these really fast 3D graphics cards in the computers on our desks and it would be a shame not to use them. But since the graphics hardware now is drawing all the pixels, I have no easy way of knowing what was drawn at any individual screen location.

The point below the mouse on the screen can really be thought of as a line extending from the mouse into the screen. To find out where it hits, I could cast a ray along that line and see where it hits on the surface of the model. This problem is very well studied. It's exactly like what is done in a raytracing renderer. However, that would mean I would need the coordinates of the transformed model in the screen. On a system that handles hardware transformation, such as Nvidia's GeForce 256, I would need to transform everything myself in software or use something like the feedback mechanism in OpenGL, which is notoriously slow. Another problem is that polygons can overlap on a model. The graphics card uses the Z-buffer to sort this out but I would need to use another sorting mechanism to determine which polygon was closest under the mouse.

In order to use the graphics hardware more effectively, I am going to use a technique known as the selection buffer (or item buffer). The idea is to use a rendered image to tell me what each pixel in my scene represents. Let me start with the simple case of a single object rendered with a 256×256-pixel texture applied. At every vertex there is a UV texture coordinate that describes how the texture maps to that vertex. If I render the object out, I get the image in Figure 4.

To get this image, I have submitted the texture coordinates and vertex positions to the hardware in a pretty standard manner, shown in OpenGL in Listing 1. Of course, I could optimize this by using more efficient graphics calls such as vertex arrays and triangle strips, but you get the idea.

For the 3D paint system, the triangle that is under the mouse is not actually important. The information I really want to know is the texture coordinates that are under the mouse. So, suppose I change the rendering loop to what I have in Listing 2.

You can see that I have only made a

very minor modification. Instead of using the texture coordinates in the normal way, I have put the texture coordinates into a `glColor3f` command so that the U coordinate is in the red position and the V coordinate is in the green position. I set blue to be 1.0 but that really doesn't matter. When I render with this routine, I get the image you see in Figure 5.

As you can see, the colors blend smoothly across the object in a nice pastel manner. What's important, though, is that each pixel of the image stores the texture coordinates that would be drawn there. The vertex color interpolator has interpolated the UV coordinates for me. So now, in order to know what part of the texture map is drawn on a certain part of the screen, I just look at the color at that point. For example, in the sample image, I can pick a point in the center of the object and I get the color (144,165,255). That corresponds to the texture coordinate (144,165). If I change the color at that position on the texture map, it will look like I painted on the object. That is pretty easy and the visibility calculations are handled by the hardware. The hardware Z-buffer will determine for me which polygon is in front.

Implementation

To make this work for 3D painting, I will need to render both the UV selection render and the actual image render. If I had to switch back and forth between those two render modes, it would be pretty distracting for the user. However, if I use a double-buffered display where one is a display frame buffer and the other one I can draw on, this won't be a problem. The double-buffered display works by waiting until the hardware has finished rendering a frame. Then you can swap the buffers to show the results. This leads to smooth animation without flickering. To use this for my selection buffer, I can render the selection data to the back buffer and not display it.

I don't even need to render the selection buffer all the time. I only really need to render it whenever the object or camera

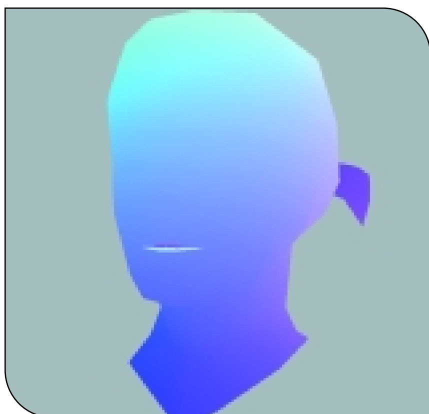


FIGURE 4 (top). A face mapped to a character model, rendered in the routine given in Listing 1. FIGURE 5 (bottom). Image with UV coordinates in the vertex colors, rendered in the routine given in Listing 2.

LISTING 1. My original method for submitting texture coordinates and vertex positions to the hardware for the image in Figure 4.

```
lEnable(GL_TEXTURE_2D);
glBindTexture( GL_TEXTURE_2D,visual->glTex);
glBegin(GL_TRIANGLES);
  face = visual->index;
  for (loop = 0; loop < visual->faceCnt; loop++,face++)
  {
    glTexCoord2f(visual->texture[face->t[0]].u,visual->texture[face->t[0]].v);
    glVertex3fv(&visual->vertex[face->v[0]].x);
    glTexCoord2f(visual->texture[face->t[1]].u,visual->texture[face->t[1]].v);
    glVertex3fv(&visual->vertex[face->v[1]].x);
    glTexCoord2f(visual->texture[face->t[2]].u,visual->texture[face->t[2]].v);
    glVertex3fv(&visual->vertex[face->v[2]].x);
}
glEnd();
glDisable(GL_TEXTURE_2D);
```

LISTING 2. Changing the rendering routine to find texture coordinates with `glColor3f`, which produces the image in Figure 5.

```
glEnable(GL_TEXTURE_2D);
glBindTexture( GL_TEXTURE_2D,visual->glTex);
glBegin(GL_TRIANGLES);
  face = visual->index;
  for (loop = 0; loop < visual->faceCnt; loop++,face++)
  {
    glColor3f(visual->texture[face->t[0]].u,visual->texture[face->t[0]].v,1.0f);
    glVertex3fv(&visual->vertex[face->v[0]].x);
    glColor3f(visual->texture[face->t[1]].u,visual->texture[face->t[1]].v,1.0f);
    glVertex3fv(&visual->vertex[face->v[1]].x);
    glColor3f(visual->texture[face->t[2]].u,visual->texture[face->t[2]].v,1.0f);
    glVertex3fv(&visual->vertex[face->v[2]].x);
}
glEnd();
glDisable(GL_TEXTURE_2D);
```

moves, showing another part of the object. I can then grab the back buffer and store it for all my paint operations. I use two easy OpenGL functions to get this done:

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

This tells the system that I want to read from the back buffer and then grabs the data and puts it in a storage buffer that I allocate to hold this information. That's all there is to it.

It's important to know that the selection buffer needs to be rendered with antialiasing and all other blending and smoothing effects turned off. Otherwise, there will be strange UV values in the selection buffer when the hardware blends it with the background.

This method is also limited to texture maps that are 256×256. That is because I only used 8 bits to store the U and V coordinates. If I want to support textures larger than this, I could use 12 bits for each coordinate with some extra manipulation, which would allow textures up to 4096×4096. That should be plenty for most applications.

What Can I Do?

Now that the system is in place, I can start painting the models in my game engine. The simplest thing I can do is replace the pixel in the texture map with a new color. However, now we are in 2D space and all kinds of 2D tricks like those in Photoshop can be used. This is where it would be nice to create all sorts of brushes of various sizes that would color the pixels surrounding the click point.

There are several ways to do this. I could assume that surrounding pixels were also surrounding on the texture map. That is clearly the easiest thing to do and is in fact what I have done in my sample application. However, that may not be the right thing to do in some cases. For example, the texture coordinates can be mapped in a nonlinear manner all over the texture map. If I choose a big brush and just start drawing around, I may overdraw the borders and bleed into another part of the object.

The other method would be to sample around on the selection buffer and draw on those points in the texture. However, depending on the object, this may cause you to miss pixels due to perspective and scaling effects. Perhaps interpolating between sample points would help this problem. The method that works the best will probably depend on the application.

I could also assign a blending value or opacity to the paint color that would decide how it blends with what is already in the texture map. Implementing something like Photoshop's airbrush tool shouldn't be too big a challenge.

At this point, I still haven't addressed quite a few issues. So far, I have assumed that the model already has a texture map assigned to it and texture coordinates are set. To make the tool more useful, I will need to add in standard mapping algorithms like planar, cylindrical, and spherical. I also want to be able to support multi-texture rendering as well as special effects such as bump maps.

This program is the first step toward creating a tool that will enable artists to use 3D technology in more interactive ways. My goal is to help artists bring their art and intelligence to the technology instead of using technology to simulate artistic methods. I am also very interested in using these techniques to modify the geometry directly. I could use displacement maps to change the model or implicit modeling techniques to add to them. I have also been intrigued with the idea of attaching painted "geometry" to an object. This technique was used with great success in the Deep Canvas system that Disney Feature Animation created for the movie *Tarzan*. But all that will have to wait until next month.

Until then, play around with my mini–3D paint program and see what kinds of effects you can create. You can download the source and application from the *Game Developer* web site at www.gdmag.com.

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

FOR MORE INFORMATION

WYSIWYG 3D PAINTING

Hanrahan, Pat, and Paul Haeberli. "Direct WYSIWYG Painting and Texturing on 3D Shapes." *Proceedings of Siggraph 1990*. pp. 215–223.

PAINTER 3D

For information go to www.corel.com.

Touched by a Vampire: Animating Physical Interactions

The interaction of actors in a movie is so common, you might not give it a second thought when you see computer-generated characters interact in a similar way. From a passionate kiss to a clawed hand grabbing a shoulder, physical interactions that establish a relationship between characters are also able to convey emotions. For the viewer they can evoke a wide range of emotional responses, from instilling feelings of warmth to frightening him out his chair. When such interactions are performed well in a movie they can increase the immersiveness and draw viewers into the story. The same rule applies in games. Players enjoy a more immersive experience since they can see and feel what characters go through.

What may be easy to accomplish in movies can be extremely time-consuming and difficult to pull off in a virtual world. For example, in Nihilistic Software's first game, VAMPIRE: THE MASQUERADE — REDEMPTION, we obviously needed vampires to suck blood. The vampires had to be able to grab an unwary passer-by or enemy, close their mouth on the victim's neck, and suck until they were interrupted or released their victim. When performing this action, the movement needed to convey the strength and quickness that one would expect from a vampire. The movements also needed to convey intimacy when the victim and vampire embraced. Besides incorporating these characteristics into the motion of the characters, they also needed to actually touch each other and react to each other. For example, when the vampire reaches to grab the shoulder of his victim, she lowers her shoulder under the weight of the vampire. Her body reacts to being pulled backward under the pressure of the arm against her waist. She shows fear until the fangs close on her neck. All these elements combine in a believable manner to help immerse the player in the scenario. Since everybody is inherently familiar with the typical dynamics of human interaction, the player might only notice what looks wrong rather than what looks natural.

The First Bite

My first attempt at a character-interaction animation was for Christof, VAMPIRE's main character, to feed on one of the civilians walking around the town, which you can see in Figure 1. Christof had to grab her, hold her and suck blood from her neck, and then release her. Before starting to work in my 3D animation program, first I had to figure out the exact dynamics of this interaction. Where do you grab someone from behind? How do you bring the victim close to you and how do you get to the neck as quickly as possible? (A vampire has presumably done this many times before so he should be efficient.) I enlisted a colleague to help me construct such an event and after acting out the scene (without drinking any of my test subject's blood), I had an idea of

FIGURE 1. An animation sequence showing the carefully orchestrated interaction of Christof and a helpless milkmaid.

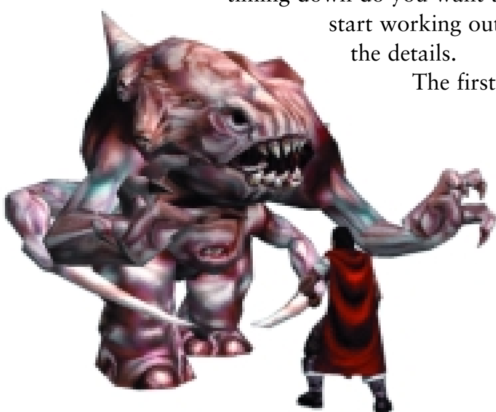
MAARTEN KRAAIJVANGER | *Maarten is the lead artist at Nihilistic Software, which just completed its first game, VAMPIRE THE MASQUERADE — REDEMPTION. He is currently busy adding more talent to the art team and taking karate classes so he can animate them for his next project. Contact him him at maarten@nihilistic.com.*

FIGURE 2. Despite differences in size between the four skeleton types, there were situations where all would need to interact with one another interchangeably in the animations. From left, Samuel, Erik, Christof, and Serena.

the basic movements and interactions between the characters I wanted. I then brought both characters into the animation program and started framing out the motion. Rather than just starting to animate from frame one forward, it's a good idea to work out some of the key positions first.

Since the interaction is important, you need to worry how about both characters should look during the animation. I like to pose the key action positions: first, the lunge where Christof has his arms extended and the milkmaid reacts with fright and starts to run away; second, the pull as he reins her in and prepares her for the final position where he has her draped helplessly in his arms.

Once the main keyframes are posed, I like to work out the timing, determining just how quick should the motions be. Going through the rough keyframes gives you a good idea of whether the timing is right. Only when you have the correct timing down do you want to start working out the details.



The first attempt I made at a feeding was very straightforward. I moved the victim forward on the Y plane and then imported Christof to the origin. (All our characters were positioned facing +Y with +X on the model's right side and +Z being up. Don't forget to make sure you are on the same page as the programmers and the 3D world they are creating, or it could get really painful for either the artist or the programmer — usually the artist loses.)

With both characters on screen I started animating Christof grabbing the helpless milkmaid and performing the embrace. I used IK chains to anchor the feet on the ground and hand-animated the interaction. With the animations complete, I exported the Christof animation as-is, and exported the milkmaid animation to the game after moving her back to the origin. Once both models were doing the animations in the game, we re-created the offset I had created while animating the action. The units of measurement differed between the game-world space and the 3D program space, so we played with the offset to get the proper interaction. The first part of the feeding sequence worked, but we then faced a new set of problems.

Size Matters

While the feeding sequence between Christof and the milkmaid was working perfectly, the rest of Christof's party was not accounted for. Throughout the game, Christof meets various vampires of all shapes and sizes to join him in his adventures. The problem is that each of the different-sized characters had to be able to feed on civilians as well as on one another. In all we had four radically different human skeleton types in the game that all needed to feed on each other, examples of which you can see in Figure 2. The Christof skeleton had an average build of around six feet tall. There was also an Erik skeleton, a thick seven-footer with bulging muscles, the petite Serena skeleton at about five-feet-four, and finally the Samuel skeleton which was also about seven feet tall but lanky. So even though the feeding animation worked for Christof and the small-framed milkmaid, it became obvious that the large skeleton of Erik would have to crunch down too much to work with the milkmaid animations and his head was too big for Christof's mouth to close on his neck.

So instead of squeezing big Erik down to the size of the milkmaid, I decided to redo the Christof animation feeding on another Christof skeleton. Once I completed the average-sized skeletons feeding on one another, it became much easier to animate the other skeletons in the same positions. The big Erik skeleton still had to be crunched down to fit within the grasp of the attacker, but not as much as if his neck had to be at the same height as the female skeleton. The small female skeleton had to be animated in a very upright position, but she fit perfectly within Christof's grasp. After completing all eight animations — a feeding and a victim animation for all four skeletons — I was on to the next problem.

It was very painful to figure out the

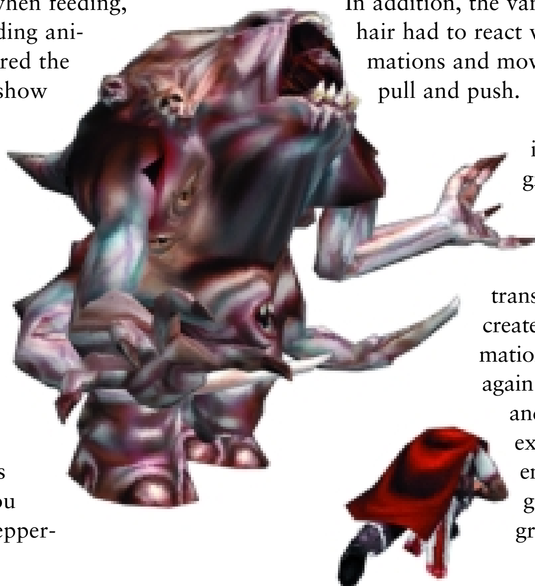
proper offsets once the animations were in the game and had to be played back. In my animation program I was moving the various skeletons in random +Y to do the animations, rather than moving each the same amount. I came up with an easy work-around by grouping the root node and moving it by a set amount. Then, after I completed animating the skeleton (the root and the joints below), I would move the model back to the origin by adjusting the node above the root. Once it was back to the origin I ungrouped the root from the extra translation node. When this technique was standardized, we no longer had any problems setting the distance for each character. Each character had the same offset and the feeding interaction worked perfectly in the game. Taking what we learned during the evolution of the interaction animations, we were then able to tackle even more complex interactions.

Awkward Match-ups

With the basic feeding animations complete, there was another set of animations we needed to tackle, as we also needed the characters to feed from each other's wrists. Wrist feeding in the game allowed the vampires under the player's control to share blood with each other and keep the player's entire party healthy. Unlike the animation described earlier, the characters had to face each other in order to wrist-feed.

Since we wanted to portray our vampires as always on the verge of losing control when feeding, the wrist feeding animation required the vampires to show distrust between each other during this vulnerable interaction. The feeder makes eye contact with his victim, then quickly looks at the other victim's wrist (like you might at a pepperoni pizza when you're really hungry), and kneels before him. The feeder grasps the victim's wrists and sinks his teeth into the vein. The victim's natural reaction is to pull his arm away, but the feeder quickly grabs hold and keeps feeding. This sucking cycle continues until disrupted by the player or automatically broken off by the party member. (They look out for themselves if you don't.) When the animation is broken, the victim jerks his arm away and gently rubs the puncture mark, but to convey his distrust he never he keeps his eye on the feeder. The feeder, on the other hand, wipes his mouth after the messy affair, and climbs back to his feet. Hopefully the player can sense both that the victim did not enjoy the experience and that the feeder just had a nice meal — hence the player's emotions are drawn into the dramatic episode.

To make these wrist-feeding animations work, I had to come up with a way for all the animations to work for all the different-sized characters. Based on what I learned from the first feeding animations, I knew I needed to start by animating the two average-sized Christof skeletons, and then animate the other three skeletons in their place. It was a bit tricky to find a good height for the wrist, since the small female skeleton became very short once she was on her knees. Since we actually had two back joints, we could bend burly Erik over enough to feed on wrists. With the constant contact, it was a tricky animation — the feeder and victim where almost playing tug-of-war with the arm. In addition, the vampires' clothing and hair had to react with secondary animations and move correctly with every pull and push.



For the wrist feeding animation I grouped the root node as before, but this time I added a rotation as well as a translation to the newly created node. After the animation was complete, I again removed the rotation and translation from the extra node so the player ends up back at the origin. When the programmers added support for 180-degree rotation in the game's animations, presto — the characters were wrist feeding on each other.

Enter the Bad Guys

At this point we were pretty confident we could animate just about any interaction between characters. So to add more intensity to the game, we started showing contact between the party characters and the bad guys. In the game it was hard to show the relative strength of the vampires and enemies, since we only had one damage animation and it didn't reflect the strength of the enemy when someone was hit. Only by using interaction animations could we show off the strength of one enemy compared with another. The first monster for which we made an interaction animation was the Vozhd. It's a massive creature about 20 feet tall that is sculpted out of the flesh of 15 to 20 humans. It stands about three times as tall as the player model and should handle a vampire like a rag doll.

I framed out the basic animation for the scene, where the Vozhd picks up a party member, lifts him up high in the air, puts him into his mouth and takes a big bite, and then finishes him off by slamming him to the ground. It was a lot of fun to play around with this animation since the Vozhd is just an unstoppable brute and the party characters are utterly helpless. In the animation (shown in Figure 3 with Christof as an example) I had the party members' arms drop by their sides like they were in awe of what was in front of them. Then, as a clawed hand comes down to grab them, they react to it by turning their head in the direction of the movement. Once the character's head focuses on the claws, they react by trying to step out of their path. Of course they are just a little too slow, and the Vozhd's claws engulf their torso. The party members initially get crushed inside the grip of the claws and then as the Vozhd raises the victim to his big open jaws, they try to kick their way out. They all look in terror at the big mouth in front of them. (Erik, always the fighter, punches the Vozhd in the head right before getting his head and arms stuffed in the creature's mouth.) Right after the party character has been

FIGURE 3. A dramatic encounter with a Vozhd creates an opportunity for subtle emotional responses from Christof.

chewed, they go limp. Their arms and legs dangle lifelessly from the claw of the Vozhd as he winds up his arm and slams them down to the ground. Depending on the health of the characters, they then either stay down for the count or get back up, ready for more.

Once More, with Feeling

Since we animated everything in VAMPIRE by hand, we had a lot of flexibility to change the animations once they were completed. Rather than have the plethora of keyframes that come with motion capture, our animations were easy to adjust since we kept our keyframe data clean. Once the main interaction animation was completed, creating the animations for the different-sized party characters went very quickly. We had so much success with it we added as many interaction animations as we could to enhance the gameplay. We created both staking and un-staking animations for all the vampires. We animated a powerful werewolf attack in which the party characters get flipped up in the air. For the final boss encounter we animated the Zulo (a giant bat-dragon monster) attack where the character is grabbed, bitten in the head, and tossed around before getting plunked back down on the ground. All of these animations enhanced the gaming experience by providing more vivid gameplay.

Immersiveness is what all games shoot for, and the relative success with which it's pulled off is what in part separates memorable games from the also-rans. The more players are drawn in, the more they will enjoy their experience. How characters interact can make a game feel more believable and provides a better player experience. Our first attempt at animating these interactions made us realize how much more is possible. Soon, interaction animations will be able to convey many more subtle emotions than what is seen in most games today. The characters in games will interact and behave increasingly like real-life actors, and this will help raise the immersiveness of the gaming experience to the next level.

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

Mission: Compressible

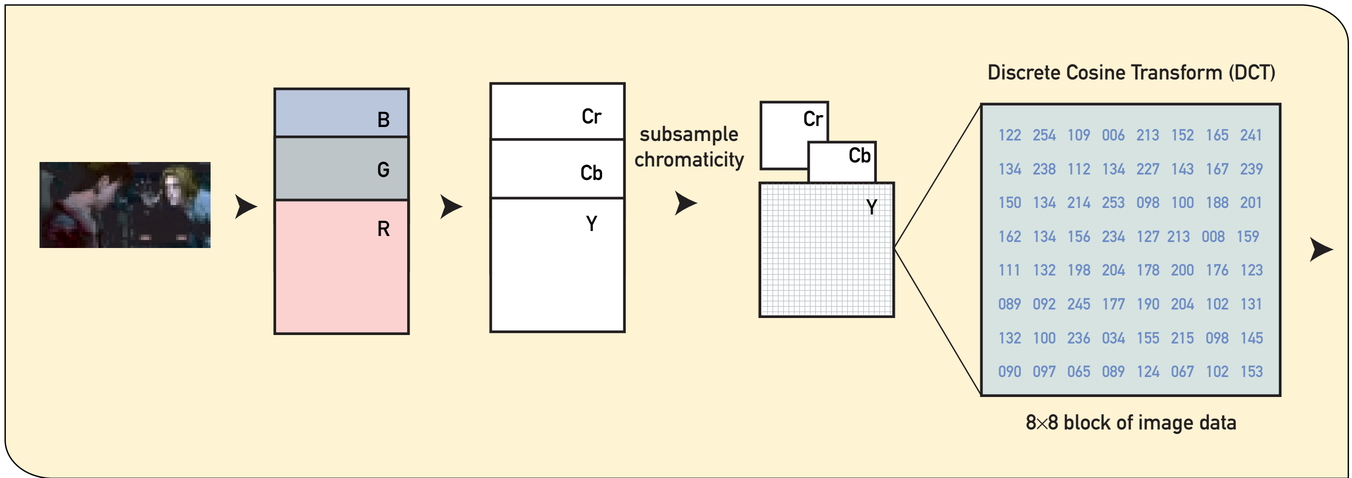
Achieving Full-Motion Video on the Nintendo 64

TODD MEYNINK | *When there's no surf to be found, Todd's busy pretending to be a software engineer at Angel Studios. Drop him a line at todd@angelstudios.com.*

RESIDENT EVIL 2 for the Nintendo 64 was the first game on a cartridge-based console system to deliver full-motion video. Angel Studios' team brought this two-CD game, comprising 1.2GB of data, to a single 64MB cartridge. A significant portion of this data was more than 15 minutes of cutscene video. Achieving this level of compression, meeting the stringent requirement of 30Hz playback, and delivering the best video quality possible was a considerable challenge.

To look at this challenge another way, let's put it into numerical perspective. The original rendered frames of the video sequences were 320×160 pixels at 24-bit color = 153,600 bytes/frame. On the Nintendo 64 RESIDENT EVIL 2's approximately 15 minutes of 30Hz video make a grand total of 15•60•30•153,600 = 4,147,200,000 bytes of uncompressed data. Our budget on the cartridge was 25,165,824 bytes, so I had to achieve a compression ratio of 165:1. Worse still, I had to share this modicum of cartridge real estate with the movie audio.

The Playstation version of RESIDENT EVIL 2 displays its video with the assistance of a proprietary MDEC chip but because the N64 has no dedicated decompression hardware, our challenge was compounded further. To better understand the magnitude of the implementation hurdles, consider that it is analogous to performing full-screen MPEG decompression at 30Hz, in software, on a CPU roughly equivalent in power to an Intel 486. Fortunately, the N64 has a programmable signal processor called an RSP that has the ability to run in parallel with the CPU.



TYPE	USED AS REFERENCE?	TYPE OF PREDICTION	COMMENTS
Intrapictures (I-frames)	Yes	N/A	Typically encoded with a JPEG-like algorithm. They start and end each GOP.
Predicted-pictures (P-frames)	Yes	P-frames support forward prediction from a previous I-frame.	The motion-compensated prediction errors are DCT coded.
Bidirectional (interpolated) pictures (B-frames)	No	A B-frame is a forward, backward or bidirectional picture created by, and relative to, other I and P frames.	

FIGURE 1 (top). A summary of the encoding process.
FIGURE 2 (bottom). GOP encoding methods.

A Brief JPEG Primer

In order to simplify the timing and synchronization issues, I chose an MPEG-1-style (henceforth referred to as MPEG) compression scheme for the video content only. (Audio was handled separately, which I'll discuss later in this article.)

As an introduction to the relatively complex issues of applying MPEG compression to the video sequences of the game, let me present a brief primer on JPEG compression.

First, the image is converted from RGB into YCbCr. This process converts the RGB information into luminance information (Y) and chromaticity (Cb and Cr):

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.2989 & 0.5866 & 0.1145 \\ 0.1687 & -0.3312 & 0.5 \\ 0.5 & -0.4183 & -0.0816 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Inverting the coefficient matrix and applying it to YCbCr finds the inverse transformation. This color model exploits properties of our visual system. Since the human eye is more sensitive to changes in luminance than color, I could devote more of the bandwidth to represent Y than Cb and Cr. In fact, I can halve the size of the image with no perceptible loss in image quality by storing only the nonweighted average of each 2×2-pixel block of chromaticity information. This way the Cb and Cr information is reduced to 25 percent of its original size. If each of the three components (Y, Cb, Cr) represented 1/3 of the original picture information, the subsampled version now adds up to 1/3 + 1/12 + 1/12 = 1/2 the original size.

Second, each component is broken up into blocks of 8×8 pixels. Each 8×8 block can be represented by 64-point values denoted by this set:

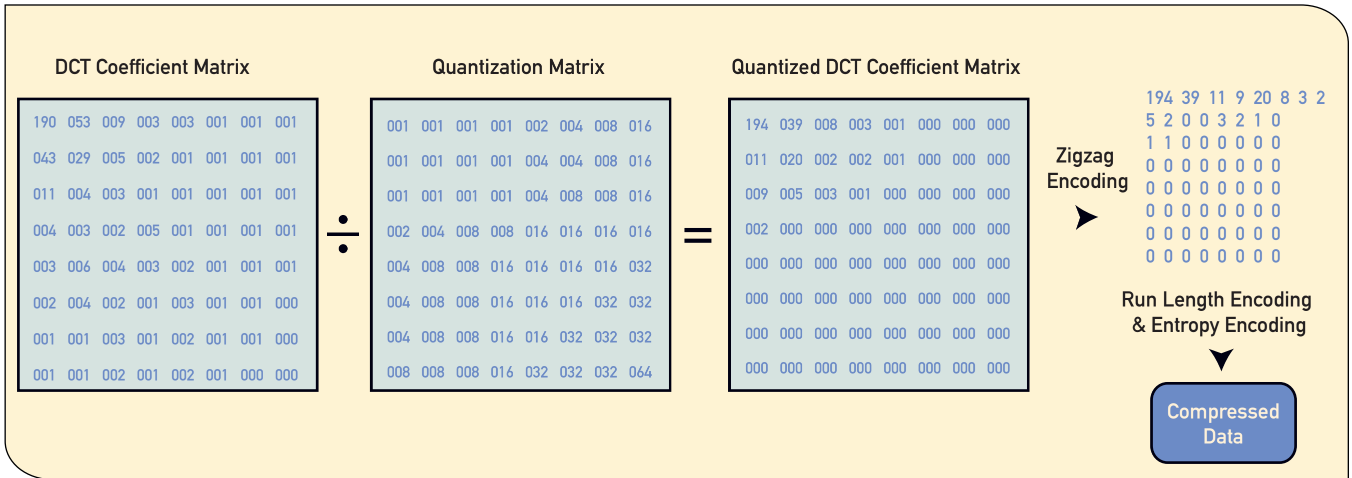
$$\{f(x,y) \in 0 \le x \le 7,\ 0 \le y \le 7\}$$

where x and y are the two spatial dimensions. The discrete cosine transform (DCT) transforms these values to the frequency domain as $c = g(F_u,F_v)$, where c is the coefficient and F_u and F_v are the respective spatial frequencies for each direction:

$$\frac{1}{4} \cdot C(u) \cdot C(v) \cdot \sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y) \cdot \cos\left(\frac{(2x+1)\cdot u \cdot \pi}{16}\right) \cdot \cos\left(\frac{(2y+1)\cdot v \cdot \pi}{16}\right)$$

$$\text{where } C(u) = C(v) = \frac{1}{\sqrt{2}} \text{ for } u,\ v = 0 \text{ and } 1 \text{ otherwise}$$

The output of this equation gives another set of 64 values known as the DCT coefficients, which is the value of a particular frequency — no longer the amplitude of the signal at the sampled position (x,y). The coefficient corresponding to vector (0,0) is the DC coefficient (the DCT coefficient for which the frequency is zero in both dimensions) and the rest are the AC coefficients (DCT coefficients for which the frequency is nonzero in one or both dimensions). Because sample values typically vary gradually from point to point across an image, the DCT processing compresses data by concentrating most of the signal in the lower values of the (u,v) space.



For a typical 8×8 sample block, many — if not all — of the (u,v) pairs have zero or near-zero coefficients and therefore need not be encoded. This fact is exploited with run-length encoding.

Next, the 64 outputted values from the DCT are quantized on a per-element basis with an 8×8 quantization matrix. The quantization compresses the data even further by representing DCT coefficients with precision no greater than is necessary to achieve the desired image quality. This tunable level of precision is what you modify when you move the JPEG compression slider up and down in Photoshop when you save an image.

In the third step (ignoring the detail that the DC components are difference-encoded), all of the quantified coefficients are ordered into a "zigzag" sequence. Since most of the information in a typical 8×8 block is stored in the top-left corner, this approach maximizes the effectiveness of the subsequent run-length encoding step. Then the data from all blocks is encoded with a Huffman or arithmetic scheme. Figure 1 summarizes this encoding process.

Both JPEG and MPEG are "lossy" compression schemes, meaning that the original image can never be reproduced exactly after being compressed. Information is lost during JPEG compression at several points: chromaticity subsampling, quantization, and floating-point inaccuracy during the DCT.

Motion Picture Compression

JPEG compression attempts to reduce the spatial redundancy in a single still image. In contrast to a single frame, video consists of a stream of images (frames) arriving at a constant rate (typically 30Hz). If you examine consecutive frames in a movie, you'll generally find that not much changes from one frame to the next. MPEG exploits this temporal redundancy across frames, as well as spatial redundancy within a frame.

To deal with temporal redundancy, MPEG divides the frames up into groups, each referred to as a "group of pictures," or GOP. The size of the GOP has a direct effect on the quality of the compressed images and the degree of compression. A GOP's size represents one of the many trade-offs inherent to this process. If the GOP is too small, not all the temporal redundancy will be eliminated. On the other hand, if it is too large, images at the start of the GOP will look substantially different from images toward the end (imagine a scene change partway through), which will adversely affect the quality of reconstructed images.

To improve compression, frames are often represented by composing chunks of nearby "reference" frames. The frames within a GOP are generally encoded via one of three methods, as shown in Figure 2.

Figure 3 shows the different frames and their roles and relationships. This example introduces an intracoded picture (I-frame) every eight frames. The sequence of intrapictures (I), predicted pictures (P), and bidirectional pictures (B) is IBBBPBBBI. When GOP size is varied, only the number of B-frames on either side of the P-frame ever changes. Note that this sequence represents the playback sequence and is not necessarily the order in which frames are stored. Storing frames 1,2,3,4,5,6,7,8 as 1,5,2,3,4,9,6,7,8 would make sense since the I- and P-frames are read first, facilitating construction of B-frames as soon as possible and reducing the number of frames that need to be kept around in order to decode the stream successfully.

Prediction and interpolation are employed in a technique

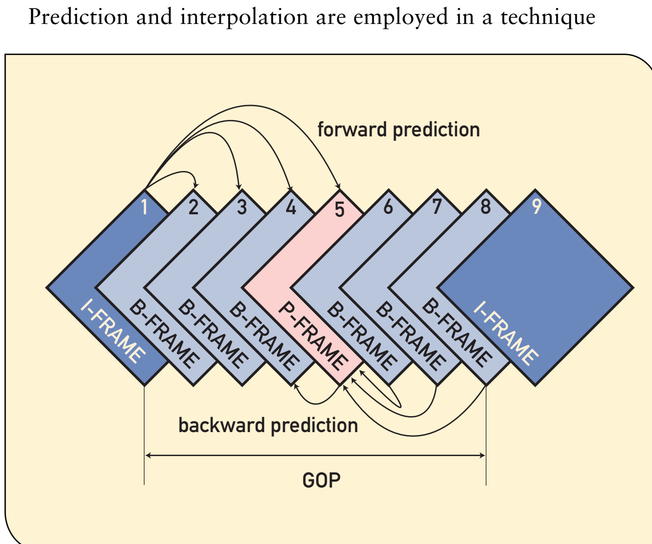


FIGURE 3. Relationships between the I-, B-, and P-frames.

called "motion compensation." Prediction assumes that the current picture can be modeled as a transformation of the picture at some previous time. Interpolated pictures work similarly with past and future references, combined with a correction term.

It makes no sense to use an entire image to model motion within a frame, so modeling motion must be done with smaller blocks. MPEG uses a macro-block consisting of 16×16 pixels (think of a macro-block as four of our 8×8 DCT blocks). This approach illustrates another trade-off between the quality of the image and the amount of information needed to represent the image. An estimate of per-pixel motion would look best, but would be way too big, while a quarter-image block would look pretty ordinary but take up very little space. In a bidirectionally coded picture, each 16×16 macro-block can be of type intra, forward predicted, backward predicted, or average. Note that the 16×16 block used for compensation need not lie on a 16×16-pixel boundary.

A cost function typically evaluates which macro block(s) from which image represents the current block in the current image. This cost function measures the mismatch between a block and each predictor candidate. Clearly an exhaustive search, in which all possible motion vectors are considered, would give the best result, but that would be extremely expensive computationally. Figure 4 shows the relative sizes of the different frame types.

JPEG Compression/ Decompression Steps

JPEG Compression

1. Preparation of data blocks (RGB→YCbCr)
2. Source encoding
   - Discrete cosine transform (DCT)
   - Quantization
3. Entropy encoding
   - Run-length encoding
   - Huffman or arithmetic coding

JPEG Decompression

1. Entropy decoding
   - Huffman or arithmetic coding
   - Run-length coding
2. Source decoding
   - Dequantization
   - Inverse discrete cosine transform (IDCT)
3. Preparation for display (YCbCr→RGB)

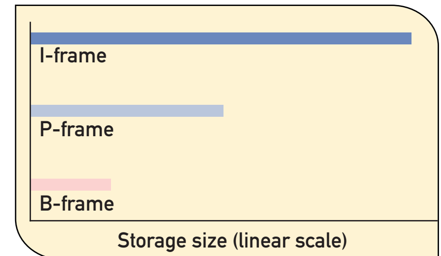


FIGURE 4. Relative sizes between I-, B-, and P-frames.

Implementation

My first step to implement the full-motion video for the N64 version of RESIDENT EVIL 2 was to develop a PC-based compression/decompression platform that could be debugged and tuned easily. This let me experiment with different GOP sizes, bit rates, and other variables. It quickly became apparent that this optimization challenge would be a war between image size, image quality, and decoding complexity.

There were severe memory constraints. As you probably know, without an expansion pack, the N64 has only 4MB of RAM. This memory is divided up among program code, heap, stack, textures, frame buffers, the Z-buffer, and so on. For a large game, it's likely that there will be room for only two frame buffers at any reasonable resolution and color depth. Keep in mind also that you need space to hold the necessary reference frames (I-frames and P-frames) in memory to compute the predicted frames. This requirement came down to three frames (I,P,I) of YCbCr data at 24-bit color. Obviously the resolution of the video dictates exactly how much RAM this requires.

I tested many different parameter settings, the most fundamental of which was bit rate. A higher bit rate naturally led to higher quality. Unfortunately, simply raising the bit rate to a point where acceptable quality was exhibited across the board required too much storage space. In our case, a quick calculation gives us our target mean compressed frame size: 25,165,824 bytes / 27,000 frames = 932 bytes per frame.

Higher resolution improved the image quality up to a point, but it quickly fell off after that. The reason for the falloff is that only a limited number of bits are available to describe all the pixels in a frame. While a high-resolution movie may look good when little in the scene is changing, rapid motion or a scene change may not be adequately described at the same bit rate. This artifact becomes extremely noticeable when the boundaries of motion blocks are discontinuous, which gives the movie a "blocky" look. Additionally, increasing the resolution means more macro-blocks, more inverse DCTs, more motion compensation, more color space conversion, and generally more decoding time. It quickly became apparent to us that displaying movies encoded at the source resolution would not be possible at 30Hz.

Since movies would be displayed at a lower resolution than the source, we needed a mechanism for scaling the decoded frames back up to full-screen resolution. We tried pixel doubling, but the results were unsatisfactory even on an NTSC screen (which hides a lot of the larger "pixel" definition). Next I tried using the N64's `rectcopy` routine (part of the N64's software library) with bilinear interpolation. This approach gave better results and remained in place until I tried a custom microcode routine — which in turn gave way to a reduced screen resolution, which the RDP scaled up automatically for free. Reduced resolution also gave the added bonus of reducing memory requirements for the frame buffers.

I tried decoding the movies to both 16-bit RGB and 32-bit RGBA frame buffers. The 32-bit image gave superior results, especially across gradations of color, though at the time the performance hit didn't justify the extra memory and processing requirements. The target color depth had several implications. Foremost were the increased memory requirements of the frame buffers. At the time I was evaluating this approach, running at source resolution and color depth would not have been pos-

Full-motion video stills from RESIDENT EVIL 2 for N64.

sible given the memory constraints. A secondary implication was the increase in computing time required to process the larger frames, further hampered by the N64's less-than-stellar memory performance. At this point, I had movies running at low resolution at 30Hz and roughly within the size requirements, but the image quality left a lot to be desired and this problem needed to be addressed. I began to think about optimization.

Rewriting the Algorithm in Microcode

My decompression algorithm was written in C, and its computation time was spread over a large portion of the source. I was not going to reap large benefits from optimizing code with MIPS Assembly without a Herculean effort and far more time than I had. While I had never dealt with the N64's signal processor (the RSP) before, I knew that its vector nature and potential to run in parallel held the keys to improved performance. (For a walk-through of the process of calling microcode programs, DMA-ing data in and out of micro-memory, passing arguments, and more, refer to Mark DeLoura's article, "Putting Curved Surfaces to Work on the Nintendo 64," November 1999.)

After getting a simple "add 2 to this number" function to work, I began to port portions of the C-based decoding code over to microcode. This task was by no means simple. The only avenue for debugging the microcode was to crash the RSP at various places and read the data cache to verify that things up to that particular point were working correctly. This process was very laborious.

A direct result of the difficulties of developing microcode was that I would only have time to rewrite a finite number of routines. A fixed-point rewrite of the inverse discrete cosine transform seemed like an obvious choice. After several painstaking days of coding and verifying this routine, it was ready for prime time. Unfortunately, the rewrite actually caused the routine to perform more slowly. My investigation revealed that a cache issue was causing this problem. As each block of pixel data is read and prepped for decode, it becomes resident in the CPU's data cache. For the RSP to process it, the data must be DMA'd from main memory to the RSP's DMEM. After processing, it must then be DMA'd back into main memory. The CPU's cache doesn't know that this potentially asynchronous process has modified the data, so those cache lines must be "invalidated" and reread to ensure that the CPU is operating on up-to-date data. The bottom line was that all this extra memory thrashing was swamping any benefit gained by the efficiency of the RSP's SIMD instructions.

My next stop was the motion compensation code. Unfortunately, the amount of code required to handle all the different kinds of motion compensation was prohibitive. The RSP's lack of a `shift` instruction didn't promise a clean implementation. Clearly, the code which finally brought the decompressed image to the screen (without further CPU intervention) stood to gain the most benefit.

Rewriting the color-space conversion (CSC) routines to take advantage of the RSP's vector architecture proved to be quite successful. The RSP was uniquely suited to this sort of task. Once the RSP had performed the conversion, the RGB data could be DMA'd from DMEM directly to the frame buffer, avoiding the earlier caching problems. This bought a noticeable performance increase and provided a corresponding increase in image quality, but I was still a long way from the quality of the original FMV.

The Epiphany

At this point, my implementation was getting closer to my goal, but problems remained. First, the image quality was still not as good as I had hoped. Second, the data files required to support this inadequate quality were already substantially over their size budget. And finally, the decoding still took too long and I couldn't see an easy way to improve it — especially since I was trying to also reduce the bit rate.

Then the idea struck me: what if I skipped every other frame and interpolated at run time? I knew if I could get this approach to work, it would simultaneously

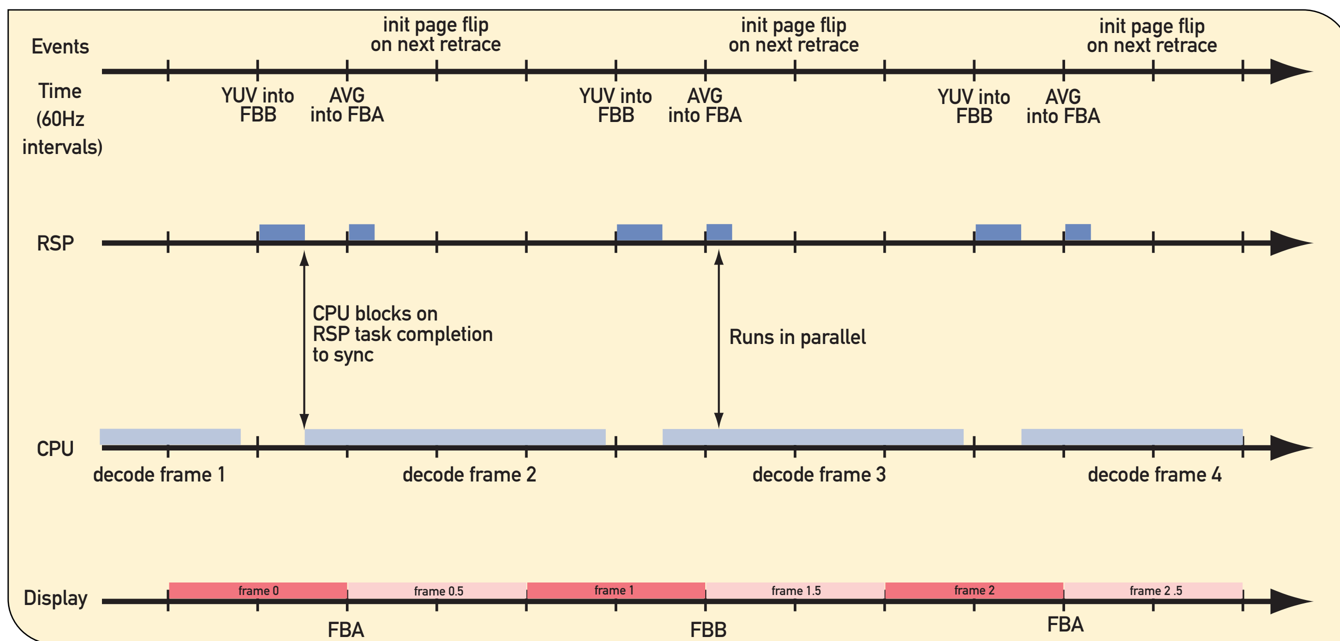


FIGURE 5. A processing time line.

halve the bit rate and double the decoding time. I was banking on the hope that it would be difficult to differentiate data decoded at 30Hz from data decoded at 15Hz with interpolation.

At first I considered using triple buffering to decode two frames, and then interpolating between the two to generate the intermediate frame. But memory restrictions quickly ruled out this approach and any of its variants.

I eventually found the solution. In it, the RSP average routine effectively swaps in a new frame without a page flip by beating the retrace gun down to the bottom of the screen thanks to some fast microcode. From a conceptual standpoint, this tricky timing allowed me to achieve triple buffering with only two buffers. (See Figure 5, a processing time line, and Figure 6, a UML state machine that runs the CPU thread in parallel with the RSP.)

With this approach, each frame had almost 1/15th of a second to decode. Skipping every other frame halved the memory footprint. This made the inclusion of all the clips possible, and also allowed us to improve the quality with the space left over. And we still had extra decoding time to burn, which we put to good use by increasing the movie resolution to further improve image quality.

It wouldn't have been possible to implement this solution without a scheduler. The scheduler used was part of a sophisticated operating system written by fellow team members Chris Fodor and Jamie Briant. In addition to supporting multiprocessing and multi-threading, it provided detailed information about and management of the N64's hardware. This was pivotal to taking full advantage of the machine. Once I fleshed out the algorithm, implementation with the OS's scheduler was straightforward.

Continuous Improvement

Shortly after we implemented this system, we created a demo for E3 1999. It was very gratifying to walk past Capcom's booth and hear people arguing over whether they were playing the game on an N64 or a Playstation. Unfortunately, the video quality on the N64 was still noticeably below that of the original Playstation game.

One of the reasons for this was that smooth color gradients were not reproducing well. I experimented with a cheap form of dithering as a postprocessing step. (Credit goes to Alex Ehrath, my fellow RE2-N64 programmer, for this idea.) As YCbCr data was converted to 16-bit RGB, I kept track of the lower-order bits that were being masked off, added these lower-order bits to the following pixel before it was masked off, and so on. The red, green, and blue channels were processed independently. While this technique provided a noticeable improvement when the frames were considered in isolation, differences from frame to frame made it look as if there were some sort of static interference when they were played as a movie. The modulation of the interpolated frames only amplified this problem.

The bad reproduction of gradients was especially noticeable in dark areas. To compensate for this, I experimented with gamma correction as a preprocessing step prior to encoding. My goal was to even out the perceived difference in intensity between dark colors and lighter colors. Unfortunately, this approach just gave the movies a washed-out look.

Next, I tackled the age-old challenge of trying to make the image on the NTSC display resemble those shown on an RGB monitor. We drew ten vertical bars across the screen, moving from black to gray to white as an intensity reference image. On an NTSC screen, the middle bar looked more red than gray, even on expensive reference monitors. After several iterations, we moved to Photoshop and applied a combination of color boosting, contrast/brightness adjust-

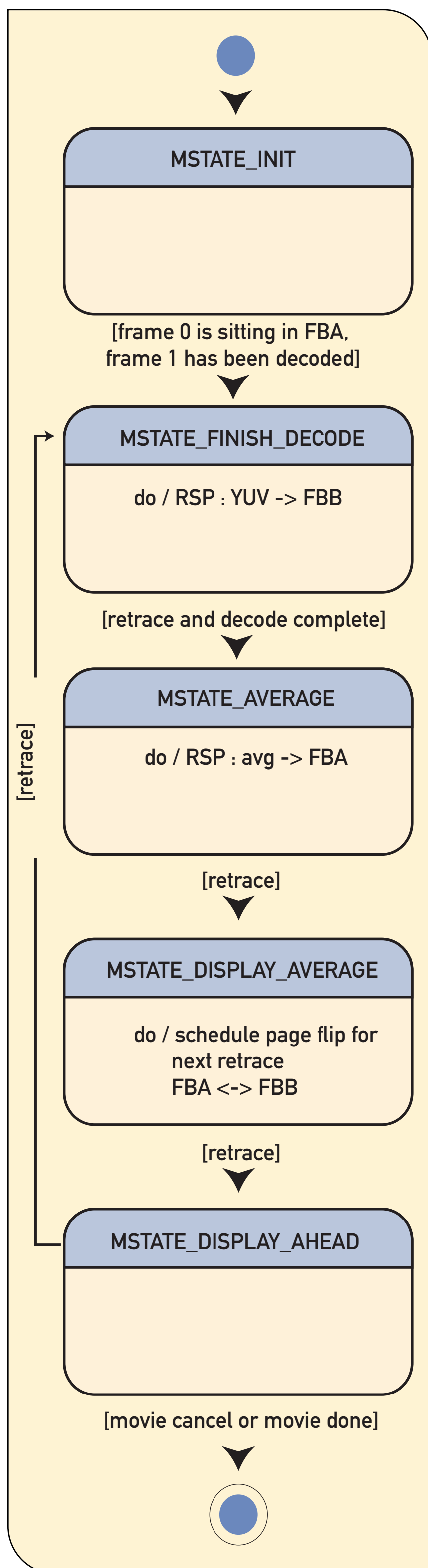


FIGURE 6. A UML state machine that runs the CPU thread in parallel with the RSP.

ments and level altering images prior to encoding until we felt we had a combination that improved the final image quality substantially.

Finally, in another attempt to improve color gradient reproduction, I retried a previously rejected technique. In earlier tests, the full 24-bit color output had looked marginally better, but extra computation and memory requirements had ruled it out. Now that the color space conversion had been moved to microcode, and a multi-processing approach had bought us much longer decoding times, I could get 24-bit color with little extra cost. A single day's coding brought startling results, and when combined with the improved color from the Photoshop preprocessing, the true-color output improved the display quality dramatically. Colors were reproduced even more vibrantly and patchy blotches became smoothly transitioning gradients. At last, I had achieved what I was after. To download an archive containing the final CSC microcode, go to the *Game Developer* web site at www.gdmag.com.

Scripting and Synchronization with Audio

Fortunately, both Leon and Claire's (the two main player-characters in RE2) games shared many sections of video, which I factored out into shared "video clips." This substantial task resulted in hundreds of clips ranging in length from a minute to a second. Movie playback was then achieved by replaying a sequence of clips. The ability to "hold" on a particular frame while the frame counter ticked by provided some additional compression. These sequences of movie clips and holds were played back through scripts that bestowed a substantial amount of flexibility.

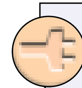
Audio compression and playback was handled separately from the video. Audio clips were triggered on particular frames.

Dividing movies into clips gave us the ability to vary the bit rate according to content. Fast action meant larger changes from frame to frame, which led to more compression artifacts requiring higher bit rates to compensate. Conversely, relatively calm scenes could be encoded at a much lower bit rate.

Changes in scenes at low bit rates were problematic when they occurred between I-frames. Until the next I-frame swung by, the sudden change caused the remainder of the GOP to display with highly noticeable compression artifacts. Quality could be preserved across changes in scene at low bit rates by making new clips with cuts on the scene change boundaries.

An Industry First

If we were to do another similar N64 project, we would definitely implement the same technology and tricks I've described here to any video sequences used. However, many of these techniques can be applied on any platform where file size is a major concern. For instance, factoring out all common "film" sequences and replaying individual clips back to back via a script to re-create the original can afford a large space savings. Ensuring the clips are built on scene-change boundaries allows you to lower the bit rate and still maintain quality. Also, compensating for loss of color saturation and levels due to compression prior to encoding can yield a result closer to the original.

Bringing full-motion video to the N64 is challenging both in terms of achieving the necessary compression to support video on a cartridge system and the software required to play the compressed data back in real time. Relentlessly trying and retrying everything brought us a great result and an industry first: high-quality video on a cartridge-based console. 

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

FOR MORE INFORMATION

BOOKS

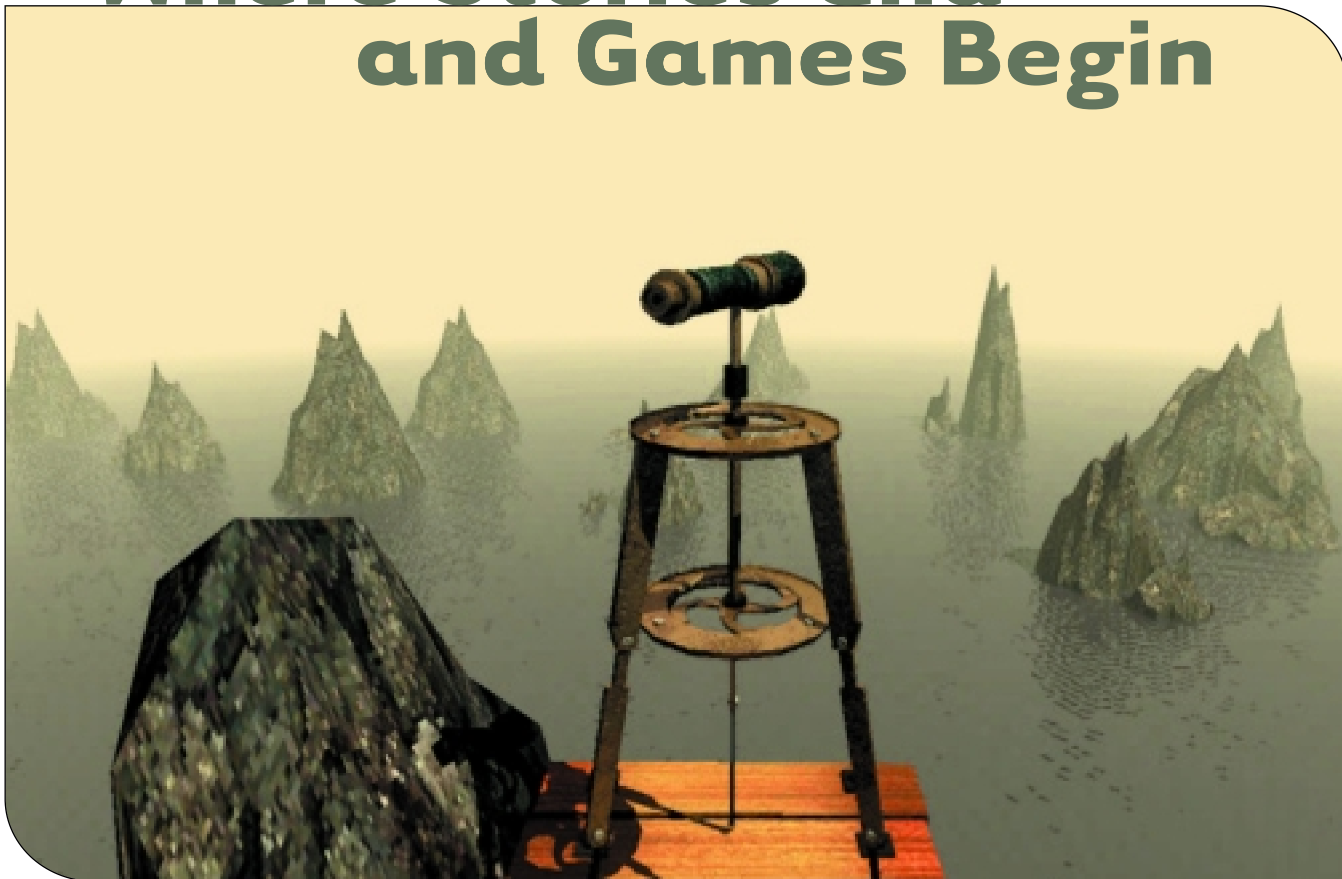
Raghavan, S. V. and S. K. Tripathi. Networked Multimedia Systems. *Upper Saddle River, N.J.: Prentice Hall, 1998.*

Foley, J. D., and others. Computer Graphics: Principles and Practice, *2nd ed. Reading, Mass.: Addison-Wesley, 1996.*

WEB RESOURCES

www.mpeg.org

Where Stories End and Games Begin



"Every medium has been used to tell stories," says Eric Goldberg, one of my oldest friends and president of Unplugged Games. "That's true of books and theater and radio drama and movies. It's true of games as well."

I have this argument all the time, and I think Goldberg's statement is balderdash. It's not true of music; music is pleasing sound, that's all. Yes, you can tell a story with music; ballads do that. So do many pop songs. Certainly some types of music — opera, ballet, musicals — are "storytelling musical forms," but music itself is not a storytelling medium. The pleasure people derive from music is not dependent on its ability to tell stories — tell me the story of the *Brandenburg Concertos*.

Nor is gaming a storytelling medium. The pleasure people derive from games is not dependent on their ability to tell stories. The idea that games have something to do with stories has such a hold on designers' and game players' imaginations that it probably can't be expunged, but it deserves at least to be challenged. Game designers need to understand that gaming is not inherently a storytelling medium any more than is music — and that this is not a flaw, that our field is not intrinsically inferior to, say, film, merely because movies are better at storytelling.

Nevertheless, there are games that tell stories — role-playing games and graphic adventures among others — and the intersection of game and story, the places where the two (often awkwardly) meet has bred a wide variety of interesting game styles. Examining them is useful, because doing so illuminates the differences between game and story — and the ways in which stories can be used to strengthen (and sometimes hinder) games.

Linearity in Games

A story is linear. The events of a story occur in the same order and in the same way each time you read (or watch or listen to) it. A story is a controlled experience; the author consciously crafts it, choosing certain events precisely, in a

GREG COSTIKYAN | *Greg has designed 27 commercially published board, role-playing, computer, and online games. He writes frequently about games, game design, and game industry business issues for the* New York Times, Wall Street Journal Interactive, Salon, *and other publications. He is co-founder and chief design officer of Unplugged Games (www.ungames.com), a venture capital startup to develop and deploy games for Internet-enabled cell phones and other wireless devices. His fourth novel,* First Contract, *was recently published by Tor Books.*

certain order, to create a story with maximum impact. If the events occurred in some other fashion, the impact of the story would be diminished (if that *isn't* the case, then the author isn't doing a good job).

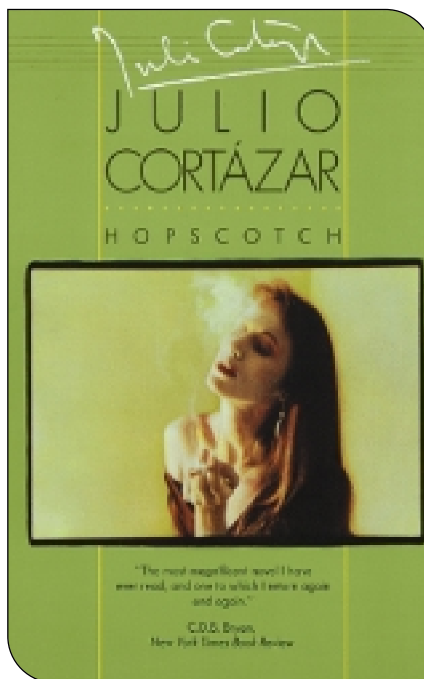
A game is nonlinear. Games must provide players with at least the illusion of free will; players must feel that they have freedom of action within the structure of the game. The structure constrains what they can do, to be sure, but they must feel they have options; if not, they are not actively engaged. Rather, they are mere passive recipients of the experience, and they're not playing anymore. They must not be constrained to a linear path of events, unchangeable in order, or they'll feel they're being railroaded through the game, that nothing they do has any impact.

In other words, there's a direct, immediate conflict between the demands of story and the demands of a game. Divergence from a story's path is likely to make for a less satisfying story; restricting a player's freedom of action is likely to make for a less satisfying game. To the degree that you make a game more like a story — a controlled, predetermined experience with events occurring as the author wishes — you make it a less effective game. To the degree that you make a story more like a game — with alternative paths and outcomes — you make it a less effective story. It's not merely that games aren't stories and vice versa; rather they are, in a sense, opposites.

Nonlinear Fiction

Maybe I'm being too restrictive by saying that stories are inherently linear. Perhaps stories have been linear to date because that's all you can do with existing media; text is read sequentially and movies are displayed as linear sequences of frames. Theater has a little more potential for interactivity but conventional theater, at least, deviates from the script only in error.

There are nonlinear forms of fiction, like Julio Cortazar's *Hopscotch*. You can read *Hopscotch* like a conventional novel, from front to back, the chapters in sequential order; or you can read the chapters in an alternative order proposed by Cortazar. Reading the book in that alternative order is a somewhat different experience. Because you encounter events and characters in a different order, the meanings of their actions are different; you see the story in a different light. Indeed, to understand the novel fully, you *need* to read it in both ways.



ABOVE. Julio Cortazar's *Hopscotch*, a notable example of nonlinear fiction.
OPPOSITE PAGE. MYST captivated audiences due in part to its engaging story.

That's great, but it's far from unique. Modern writers frequently play with the nature of narrative and time. Proust's *Remembrance of Things Past* is nonlinear in time, a sequence of remembrances as they occur to the protagonist. Joyce's *Finnegan's Wake* is filled with stream-of-consciousness nonsense words that, somehow, make sense in context. Vonnegut's *Slaughterhouse-Five* darts seemingly randomly between the decades. *Hopscotch* is creative and interesting in the way it plays with the nature of narrative, but so are many other novels.

But all of these narrative experiments are tricks. *Hopscotch's* method of presenting a narrative in two ways is interesting once, but we won't see a whole subgenre of *Hopscotch*-like novels because it's not *that* interesting. Still, *Hopscotch* gives you two paths through the same story space. The experience of each path is different and Cortazar has been clever enough to use that difference to impart somewhat different experiences. But it's only two paths. *Hopscotch* is more gamelike than a typical story but it's still a long way from a game.

Hypertext Fiction

From *Hopscotch* we move to hypertext fiction of the type promoted by Robert Coover of Brown University (see http://landow.stg.brown.edu/HTatBrown/CooverOV.html). Hypertext fiction works something like a web site; you begin by reading a bit of text, which can vary in length from a sentence to several paragraphs. Certain words or phrases are links to other bits of text. The texts of the work are linked together in a spider's web of paths. Sometimes art, video files, music, or sound is used to accompany the text, but since most of the creators of hypertext fiction come out of literary academia the focus is on words. In other words, it's not all that different from HTML, although hypertext fiction is usually implemented in a stand-alone application such as Storyspace (see www.eastgate.com). And unlike *Hopscotch*, hypertext fiction has multiple paths.

Some hypertext fictions have multiple endings, others don't have any explicit ending at all. The basic idea is that you explore the story — moving from one branch to another, gradually gaining an understanding of what's happening. The analogy to a traditional story's ending is an epiphany, which can happen whenever you've explored enough of the text — a sudden insight or "Aha!" that draws what you've read together into a coherent whole.

To the proponents of hypertext fiction, this is a completely new art form. Certainly, it is a different method of storytelling. But it is, at most times and under most circumstances, an *inferior* method of storytelling. Precisely because authors have less control over how the reader encounters their story, they cannot structure the story for maximum effectiveness. Unquestionably it is still possible to tell a story this way — but other than the novelty of storytelling in this alternative

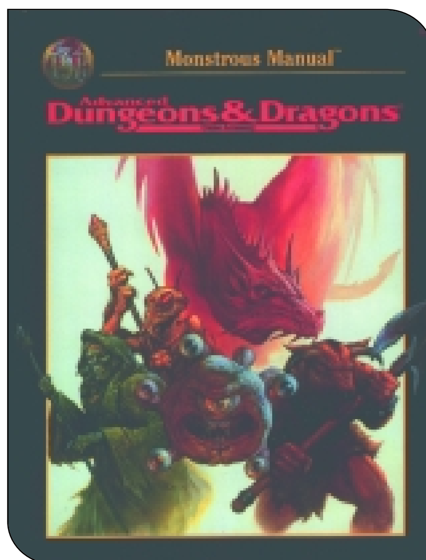
mode, there seems little reason to want to do so.

Moreover, hypertext fiction lacks one of the key ingredients that makes games compelling; there is no real goal for the reader other than getting to a point where he or she "gets" the story. You're faced with a series of decisions — follow this path or that one — but there is no context for your decision. There is no reason other than the desire to explore to choose one path over another. Reading hypertext fiction, unlike playing a game, is purposeless exploration and does not produce the same sense of desire, of compulsion, to "play." In other words, hypertext fiction is an unhappy compromise between traditional stories and games. It's gamelike in that the player has a variety of options, but not surprisingly, since it's created by people who by and large have little interest in games, it has few of the other attributes that make games appealing. Works of hypertext fiction are lousy games.

Game Books

Hypertext fiction — a highbrow, literary academic form — is closest in nature to the which-way book, a more mainstream book format, published primarily for young adults. Which-way books, also called "game books" or "choose-your-own-ending" books, had their heyday in the mid-1980s when Bantam published dozens of *Choose Your Own Adventure* books in the U.S. and the *Fighting Fantasy* game books by Steve Jackson and Ian Livingston were worldwide bestsellers (mostly outside the U.S.).

In a game book, you begin by reading the first page or two. At the end of the page, you're faced with a decision. Depending on what you decide, you turn to one page or another — if you choose option A, you might go to page 16, while option B might send you to page 86. The idea is that you're taking the role of a character, and you're trying to solve his narrative problem, whatever that may be. Some paths through the game book lead to failure, others to success. Often, "failure" means "you die, start over." This is, obviously, rather dull. Yet a work of this type has to allow players to make decisions that lead to dull stories; players of a game, of



whatever type, need to have the freedom to make decisions within the structure of the game, even if those decisions make for lousy stories.

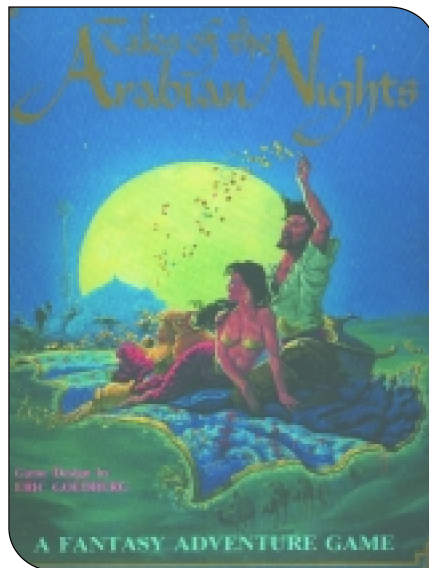
The best of these books contain some rudimentary game system to handle the resolution of, say, combat. At times, instead of simply turning the page, the book will tell you to use the game systems described elsewhere in the book to, for instance, resolve combat with a dragon whose game stats are such-and-such. This is superior to simply turning the page, because there are a range of possible outcomes, rather than single, discrete options — you go into the next combat situation with more or fewer hit points, greater or lesser skills, and so on.

Is this really all that similar to hypertext fiction? In fact, it's virtually identical; follow a link to a new bit of text. One genre is a milieu for "intellectuals," the other presumably for degraded hacks, but the essential forms are the same.

Solitaire Adventures and Paragraph-System Board Games

Game books have direct analogues in paper role-playing games and board games. The RPG analogue is the "solitaire adventure." As in a game book, the player of a solitaire adventure begins by reading a numbered paragraph or set of paragraphs, and often then turns to a different paragraph, depending on his or her decision. However, the player is also expected to be familiar with the rules of the role-playing game for which the adventure is written (Dungeons & Dragons, perhaps). Thus an external game system already exists and is used to resolve many occurrences during the course of the adventure. As a result, solo adventures are generally richer and more interesting than game books, although subject to the same basic problems.

LEFT. Dungeons & Dragons, a role-playing environment familiar to many.
BELOW. Tales of the Arabian Nights, a paragraph-system board game.



The board game analogue is the paragraph-system board game. Eric Goldberg's Tales of the Arabian Nights is the best example of this genre to date. (It is, unfortunately, long out of print.) In a paragraph-system board game, you have a piece (or pieces) on the board, and an external game system to manage its movement and other facets of the game. At various times, the game directs you to read a numbered paragraph in an accompanying booklet. That paragraph generally has you make a decision and turn to a different numbered paragraph, perhaps using aspects of the game system first. In other words, you're playing a board game which sometimes requires you to play through a mini–game book, then returns you to the overarching board game until the next mini-adventure begins. Arabian

Nights is exceptional because the system repurposes paragraphs, using the same text with different outcomes and in different ways. This provides a greater variety and more replayability than game books or solo-play adventures.

Game books also have an arcade game analogue in Don Bluth's DRAGON'S LAIR. DRAGON'S LAIR was quite popular when it was first released in 1984 because it was the first time anyone had seen cinematic-quality character animation in a videogame. Recall that in the game you play a fantasy hero who is penetrating a dragon's lair. You view a short animated sequence and then you must make a decision — you move the joystick one way or another to determine which way you want to go. At each point, there is only one "correct" path; all others end in death. To win, you have to keep on feeding quarters to the machine, dying over and over, until you can make the right decisions at each point, pretty much on autopilot.

Fundamentally, this stinks as a gameplay concept. It is frustrating and tedious to have to start over and over and maneuver through the same decisions, and subsequent games of the same style failed miserably. DRAGON'S LAIR's success was due simply to its novelty.

Text and Graphic Adventures

Text adventures are somewhat more sophisticated versions of the branching-story concept. At various locations, there are items you can pick up or interact with. Using items in certain locations and combinations opens up paths to new locations, and winning the game requires two things: solving puzzles (meaning figuring out when and where to use items to get the effects you want) and guessing what words the parser will understand (so you can get the system to do what you want).

Text adventures feel far more free-form than game books, but the same basic principle is at work here: you make decisions as to where to go and what to do. The main difference — and it is a real strength — is that the game world can respond interactively to your decisions and past actions. New paths can open up, new items become available. It's as if the text on page 86 could change in response to choices elsewhere in the game book.

Text adventures evolved into graphic adventures, which often boasted characters with whom you could talk. But again, these conversations are a matter of making discrete choices. An NPC says something, and you can respond from a menu of three or four conversational gambits. Depending on what you say, the character responds in some way, perhaps giving you another menu of things to say. Conversation is a matter of working down a conversational decision tree, and even though players may listen to voice-overs and look at screen animations, they're still working their way down a decision tree much the way game books and works of hypertext fiction operate.

Animations or video are often used in graphic adventures to provide story context; in extreme cases, as in TEX MURPHY: OVERSEER, these sequences overwhelm the game itself, making the whole seem less like a game and more like a story with minor and not terribly interesting opportunities for gameplay. When well done (as in GRIM FANDANGO), graphic adventures provide a happy combination of storytelling and puzzle-solving gameplay that holds a player's interest for hours at a time.

And yet even at their best, graphic adventures have flaws. They try to provide an illusion of player free will, but ultimately they are linear stories. A player may have freedom to move about a constrained space and solve the puzzles there in a variety of ways, but the designers control access to the next story node. Graphic adventures try to avoid branching structures that require them to create media assets that many players won't see, for the obvious reason that time and budgets are limited; if something is on the disk, you want players to encounter it. And because graphic adventures are instantial (meaning everything the player can encounter is pre-

RIGHT. Animated scene from DRAGON'S LAIR.
BELOW. GRIM FANDANGO offers players both engaging storytelling and puzzle solving.

rendered, there on the disc, and nothing is generated on the fly), they can only respond to players in ways anticipated by the designer. If the designer doesn't figure out that some players will want option X, the designer won't include that option.

All games are structures; but graphic adventures are particularly constraining structures. They're so structured precisely because they are story-dependent; they must tell good stories, and must constrain player's options and paths through the story in order to ensure that a good story is told.

Given the inherent challenges facing graphic adventures, what's amazing is the number of good games that have come out of the genre. Indeed, the best-selling computer game of all time, Rand and Robyn Miller's MYST, is a graphic adventure.

MYST: a graphic adventure and the best-selling computer game of all time.

PC and Console RPGs

From graphic adventures, we move to computer and console role-playing games such as ULTIMA and FINAL FANTASY. I don't have to explain the details of this genre to readers of *Game Developer*, so it suffices to say that these games are intimately tied to a story, but are more free-form experiences for the player. Unlike graphic adventures, the obstacles you must overcome are rarely in puzzle form; electronic RPGs aren't games of picking up things and using them to change the game state. Often obstacles are overcome by killing them, sneaking through defenses, overcoming computer security, casting magic spells, and so on.

Because player characters can vary widely in their skill sets, RPGs must be more flexible than adventure games. They must be designed so that any reasonable character can overcome the game's obstacles with a little cleverness, whatever the character's abilities. There are generally multiple solutions to problems and as a result, players feel like they have more freedom. They can approach problems in several different ways with the freedom to choose whether they'll play as a hack-and-slash, combat-oriented character, or one who prefers to be sneaky, or one who specializes in magic. Often, the player has some choice about which "space" to enter next, such whether to go into the dungeon or to the town. As in a paper RPG, character growth is important as characters gain new skills, spells, abilities, and equipment as the game progresses. Eventually, the player overcomes the final and ultimate obstacle, and fulfills his or her quest.

In other words, story is still fundamental to the electronic RPG, but the game structure allows far more freedom of action to the player than the adventure game. And the "story of the game" can differ greatly from one player to another, because the characters controlled by the players can be very different. Electronic RPGs have limited replayability, however, because the player is presented with essentially the same obstacles from game to game, and many (such as the FINAL FANTASY series) are extremely linear in nature.

Paper RPGs are similar in some ways to electronic ones but are far more free-form. The rules of the game provide a structure for resolving player actions: there are rules for combat, magic spells, advanced technology, the use of skills, and so on. Unlike electronic RPGs, there is no pre-established story line, although most paper RPG rule books contain one or several stories for new game masters to use. The expectation is that a game master will invent his or her own stories for the other players, using the rules system as needed. Alternatively, game masters can go out and buy adventure supplements, each containing a story arc which game masters can either use as written or adapt to their campaigns. Interestingly, VAMPIRE: THE MASQUERADE — REDEMPTION and the forthcoming NEVERWINTER NIGHTS both allow a player to serve as game master and craft an experience for other players. The same basic paradigm in paper RPGs is (finally) being adopted in electronic media.

Paper RPGs, unlike electronic ones, are truly social affairs. Players get together periodically to play, and spend at least as much time "role playing" — acting out — for their friends as they do trying to maximize their character's effectiveness in a purely structural context. It's common for a group of friends to get together for years on end, playing the same characters in the same game world with the same game master. In the process, they establish long character histories, flesh out the world background, and so on. For long-term role players, the stories they create through play can be as emotionally powerful and personally meaningful as anything you find in a novel or movie, and perhaps more so, because they are personally involved in their creation.

These "stories" are meaningful to players precisely because they are intimately involved. Players frequently write up "expedition reports," in which they retell the story of a particular session of play, or

several sessions. Expedition reports almost invariably make dull reading for those who are not involved in the campaign, because they do not have the same intimate familiarity with the setting and the same long history with the players and their characters. Moreover, the rhythm of a role-playing game is not the rhythm of a short story. There are peaks of excitement and periods of boredom and things happening here and there. Instead of a long build-up leading to catharsis, there is gradual character evolution. The closest noninteractive analogue is perhaps a series comic book, a comic with a small cast of characters who have adventures together, some of them short, one-issue stories, while others have story arcs that are told over several issues.

But as for the stories themselves that arise from role-playing gaming, many players never give the story a second thought. They get their kicks from solving problems and playing roles, they don't terribly mind whether the things they encounter knit together into some kind of coherent story. For them, that isn't their main interest in the game.

The Continuum Between Stories and Games

This laundry list of fiction and game genres, starting from *Hopscotch* with its single branch and leading to paper RPGs, comprises a continuum. We've moved from things people call "stories" to things that people call "games," but we've done so by moving along a spectrum of possible "game-story" genres, from ones that are close to pure story with a minimal game appendage, to ones that are close to pure game, with a residual connection to story.

The difficult decision, then, is where to place the dividing line between stories and games. Clearly, this choice is a matter of culture or taste. Because we've moved along a continuum, drawing a line somewhere would be arbitrary. As I've argued, game books and hypertext fiction are structurally identical, differing only in their implementation (print books or electronic application). Yet the culture views game books as "games" and hypertext fiction as "stories."

Hopscotch is clearly a good story; Dungeons & Dragons is clearly a good game. But even the best "stories" along our continuum — some hypertext fictions certainly qualify — have to compromise the nature of "story" in order to work. And even the best games have to compromise the nature of "game" in order to work as storytelling environments. Designing or writing here, at the intersection of story and game, is an interesting exercise, but fraught with peril and unhappy compromises.

That is true because story is the antithesis of game. The best way to tell a story is in linear form. The best way to create a game is to provide a structure within which the player has freedom of action. Creating a "storytelling game" (or a story with game elements) is attempting to square the circle, trying to invent a synthesis between the antitheses of game and story. Precisely because the two stand in opposition, the space that lies between them has produced a ferment of interesting game-story hybrids. And yet the fact remains: game and story are in opposition, and any compromise between the two must struggle to be successful.

So should designers eschew attempts to inject story into the games they design? By no means; past efforts to do so have been fruitful, and have led to interesting and successful games. What designers must do, however, is understand that they are *not* involved in the creation of stories. Gaming is not inherently a storytelling medium any more than music — just as games are not simulations (though some games are) and games are not competitions (though some games are).

To think of games as "a storytelling medium" leads to futile attempts to straitjacket games, to make them more effective stories at the expense of gameplay. Instead, designers should use story elements to strengthen their games when appropriate but not be afraid to shy away from story entirely at times. Because ultimately, what a player takes away from a game is not the story it tells (if it tells one at all), but modes of thought and ways of attacking problems, and a sense of satisfaction at mastery.

Let's look at it another way. Storytelling is fundamental to what it is to be human. Since hominids evolved the ability to speak, we've been telling each other stories. Every one of us tells stories every day; storytelling is not something that only "real" authors or "real" screenwriters can do. Every day we craft stories about the things that happen to us, and tell them to our family and friends.

A couple of days ago, I went to see my dad in the hospital. He was off the respirator, thank God, talking and awake, but still quite weak and a bit confused. He was under the misapprehension that we were all in London and going to the theater that night — he kept on asking me to make sure that I picked up the tickets. Mind you, it's probably more cheerful to think that you're going to the theater in London than to realize that you're a heart patient in an intensive care unit.

That's a story. It's a true story, but a story nonetheless. I've already told it to several people. The experience is shaped into the form of a story, to allow us to tell it, in a coherent and understandable fashion, to others.

Play is equally fundamental to what it is to be human. Infants play before they can speak and most adults play, too — with their children, with their pets, in a softball league, on poker nights. Play behavior continues to be important for learning later on in life, though most people don't think of what they're doing as "play" when they do it. When you start up a new software application, you experiment with it, try different things, see what different menu items do. That's playing. When you design a new marketing campaign, you come up with several ideas, run them past your colleagues, chat about them — you're playing with the ideas. You're experimenting with different behaviors, you're seeing what works, you're exploring the structure of the system. None of this is a game; a game is a particular, structured form of play, just as a novel is a particular, structured form of storytelling. Play is fundamental to being human, as storytelling is also, but in quite a different way.

What happens after you play? Frequently, you make up a story about what happened. When you go home to your spouse after a softball game, he or she asks how the game went, and you tell a story about the game. When your boss asks you how the plan for the marketing campaign is going, you tell him or her a story about the ideas you've experimented with so far and what your plans are for the near future. First you play; then you tell a story about it.

Play is how we learn; stories are how we integrate what we've learned, and how we teach others the things we've learned ourselves through play. But play comes first.

Evoking Emotion from Games

Chris Crawford, an important figure in the history of computer gaming and an articulate thinker about game design as an art, has said that games will never come of age until they can induce in players the same depth of emotion as a well-told story. Indeed, Crawford left computer game design to create Erasmatron (www.erasmatazz.com), an engine for creating interactive stories. He left the game industry because he believes that industry pressures have made it virtually impossible to develop worthwhile and meaningful games. Crawford now seemingly doubts the possibility that gaming can ever become a true art form.

Is Crawford right? Is it true that games will never amount to squat until they are as emotionally powerful as stories? And is "story" therefore inherently superior to "game"?

It is a mistake to assume that the value of a work of art lies solely in the emotions it engenders. Music can move us, but is emotion per se truly what we find appealing about music? Personally, I'd argue that emotion in music is tantamount to schmaltz. The classical work I prize most highly has instead a clean, almost mathematical inevitability about it. Paintings can move us, but are the canvases we regard most highly necessarily those that produce the strongest emotional response? If so, why are subdued portraits often valued more highly than monumental and busy paintings depicting momentous events? So the underlying assumption that value depends on evoking emotion is questionable.

Second, the assumption that games evoke second-rate emotional response takes into account only the emotions an artist intends to elicit from his or her audience; the sadness of the tragedy, the laughter of the comedy, the quiet serenity of a blissful piece of music. This considers only a work's inherent emotion, the emotion the creator stuffs into it. But games do engender strong emotions, such as glee, despair, frustration, satisfaction at accomplishment, and friendliness (or rage) toward other players. No game designer says, "I wish to a design a game that engenders glee in its players," but a game designer is very satisfied if he sees players of his game becoming gleeful. Precisely because games are interactive, because the player actively participates in creating the experience, the emotions evoked by a game are more organic, emerging from the interaction between game and player. Emotions cannot be drawn from game players the way they can from a theatrical audience, they cannot be stuffed into the work by the artist in the same way. Yet emotion still unquestionably exists in and is elicited by the game. Because the player's experience is at least as much his product as that of the game designer, the emotions he feels can affect him much more deeply than the

kinds of empathetic response you feel when passively viewing or reading about characters in a story.

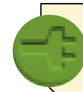
Even if we accept the assumption that an artwork's merit lies in the emotions it produces, we must reject the notion that games do not produce emotions as strongly as stories. Games do produce emotion; they simply produce different emotions in different ways.

Crawford is far from alone in abandoning games for more "important" forms, though he has chosen a different path from others who have done likewise. Chris Roberts, creator of the Wing Commander series (among the most successful computer games published), left games to direct a movie. Robyn Miller, codesigner along with his brother Rand of Myst and Riven, departed game design, also for a career in the movies. In general, many game developers fantasize about careers in film, the way that some screenwriters fantasize about careers as novelists. Why is that?

On one level, it's a status thing. Game designers view movies as more legitimate, more important than games, just as screenwriters view novels are more legitimate, more important than movie scripts. But it also has to do with the fact that movies and novels are our fundamental storytelling art forms, whereas games are the art form we created based on the fundamental human activity of play. Neither is superior to the other in any meaningful sense. To think that stories are somehow more legitimate than games is like thinking that music is somehow more legitimate than poetry, or poetry more legitimate than painting. It's comparing apples to oranges. It's the merit of the individual product within the form that matters — whether the poem is good or bad, the music soaring or trite, the game well or ill designed.

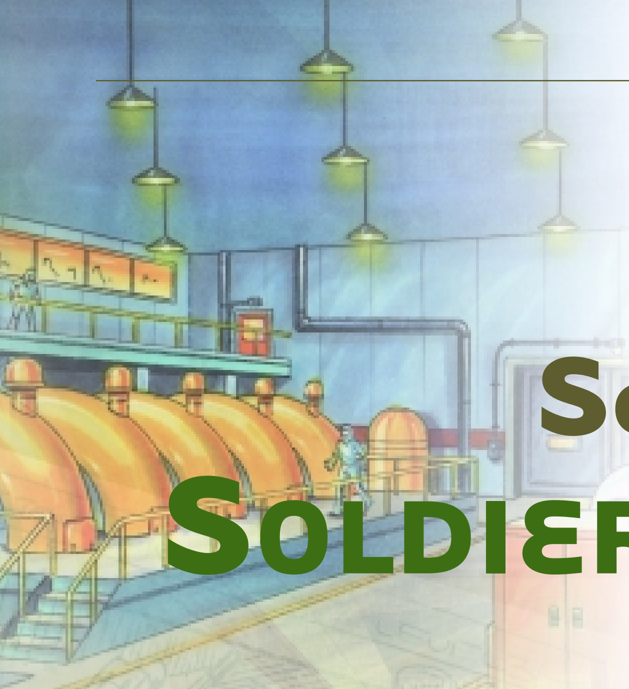
If the outside world views what we game developers do as lacking merit, the correct response is not to abandon games in an attempt at greater recognition through other media, but instead to strive to create games so well crafted, so imaginative, and so fine that their merit shines forth brightly enough that anyone can recognize their worth. The solution, in other words, is to create legitimacy for the form in which we work by creating games of enduring merit.

Gaming is the most vital art form of the age, a field that has burgeoned from virtually nothing to one of the world's most popular forms in no time flat, a field that has seen and continues to see an enormous ferment of creativity, a field that may well become the predominant art form of the 21st century, as film was of the 20th, as the novel was of the 19th. By God, we're privileged to be here at the birth of this great form, of the creation of a democratic art form for a democratic age, the creation of structures of desire, of ways to enable people to create their own entertainment through play.

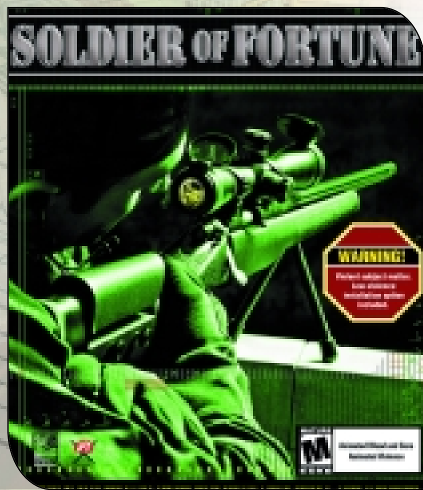


Discuss this article in Gamasutra's Connection!
www.gamasutra.com/discuss/gdmag

INT. ESCAPE SHAFT-NIGHT
- RUNNING LEGS
ACROSS SCREEN-
MOTION!
RELIEF
VALVE
DZ-121

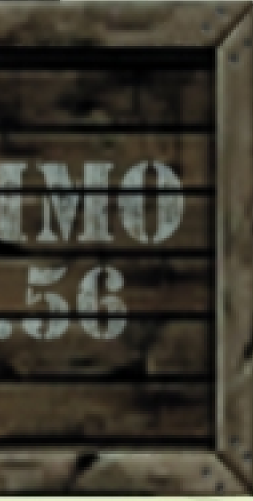


Raven Software's SOLDIER OF FORTUNE



GAME DATA

PUBLISHER: Activision
FULL-TIME DEVELOPERS: 20 (on average)
CONTRACTORS: 2
BUDGET: Multi-million-dollar budget
LENGTH OF DEVELOPMENT: 23 months
RELEASE DATE: March 2000
PLATFORMS: Windows 95/98/NT/2000, Linux
HARDWARE USED: Dell Pentium 550 with 128MB RAM, 18GB hard drive, and a TNT2
SOFTWARE USED: Microsoft Visual C++ 6.0, Microsoft Visual SourceSafe 6.0, 3D Studio Max 2.x, Softimage 3D, Photoshop
NOTABLE TECHNOLOGIES: Licensed the QUAKE 2 engine from id Software (using OpenGL), motion-capture data from House of Moves, force feedback, A3D/EAX 3D sound, World Opponent Network (WON) matchmaking services
PROJECT SIZE: 406,044 lines of code, 602 files



The development of SOLDIER OF FORTUNE was rife with questions and uncertainties right from the very beginning. Fresh from finishing up PORTAL OF PRAEVUS, the HEXEN 2 mission pack, Raven was ready to dig in to a full-fledged stand-alone product. Unfortunately, no one at Raven had a solid idea for our next project and we found ourselves floating in a sea of ideas without a solid direction. With a full team ready and willing to go, we needed a project and we needed one fast. It was then that Activision handed us the *Soldier of Fortune* license.

In the beginning, what was to become the SOF team was focusing on several different story lines and game ideas. One of these was a somewhat real-world, military-style shooter based in a World War II setting. When we decided not to pursue that game, we began looking for new game ideas. We knew that we still wanted to do a real-world military game, but beyond that we didn't have much of an idea. As soon as we got the *Soldier of Fortune* license, though, the groundwork for the game immediately began to fall into place.

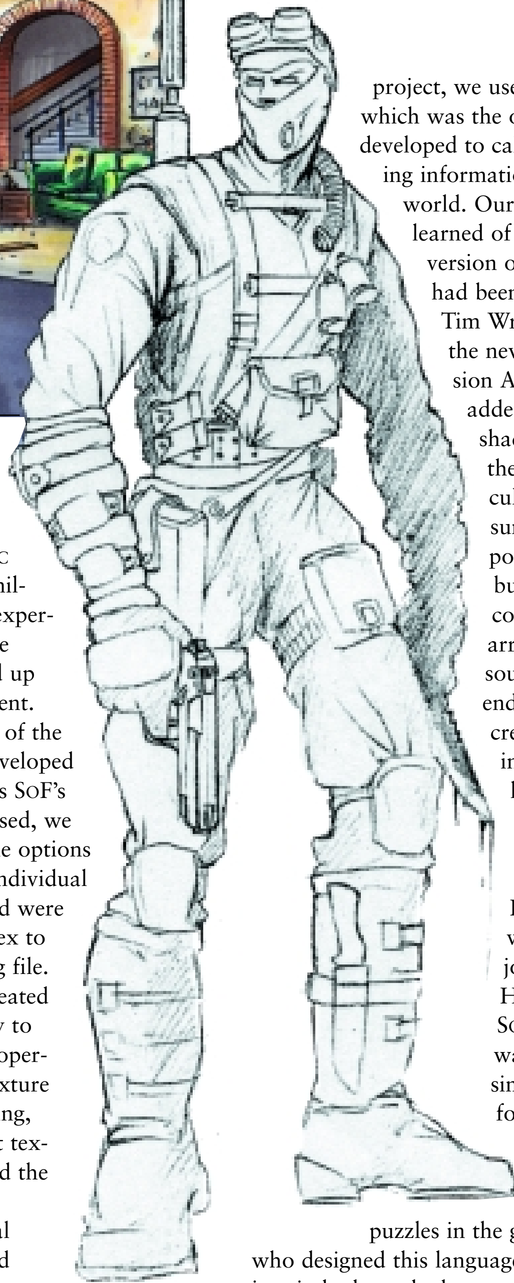
While the license name itself was met with mixed reactions from the SOF team, at its core was everything that we wanted from the game. Action, intrigue, political turmoil, and firepower were key elements of the design from the very beginning. Now we needed to find a story that would complement the license and turn it into a great game.

The name SOLDIER OF FORTUNE evokes different images for different people. One thing that we could all agree on was that the title reflected the mercenary life; making money at the risk of death. This was something that we wanted to highlight and focus on dramatically throughout the game. However, focusing on this one aspect tended to blind us to the bigger picture of what we were trying to accomplish, and our first few story attempts failed miserably. We focused too much of the gameplay on making money and not enough on finding something that would truly compel the player throughout the game. Nevertheless, even without a story set in stone we began the production of the game. This was a decision that we would come to regret many times throughout the rest of the development cycle.

The bright side to spending a large portion of development time working on a game without a solid story was that most of it was spent on technology creation. The bad part was that many of the levels that were originally planned and created had to be reworked or removed from the game entirely. On top of that, Activision was getting a little nervous that they had not seen any solid gameplay from us yet after almost a year of development. This uneasiness

ERIC BIESSMAN | *Eric is a senior designer at Raven Software and was the project coordinator for SOLDIER OF FORTUNE. Previous credits include CYCLONES, HERETIC: SHADOW OF THE SERPENT RIDERS, HEXEN, HEXEN: DEATH KINGS OF THE DARK CITADEL, HEXEN 2, and HEXEN 2: PORTAL OF PRAEVUS. Contact him at ebiessman@ravensoft.com.*

RICK JOHNSON | *Rick is a senior programmer at Raven Software and was the programming director for SOLDIER OF FORTUNE. He has been at Raven since the beginning, including the programming on Raven's first game, BLACK CRYPT. He can be contacted at rjohnson@ravensoft.com.*



ABOVE. Early sketch of the gang members' lair.
RIGHT. An early terrorist sketch.

itself caused major turmoil in the development and it took a while for us to settle into the game that we would eventually create.

Luckily, during this time, all of the core technology was implemented and functioning smoothly. Because of this, once we nailed the story down, we were able to jump head-first into the production and quickly create a solid product. In order to achieve a strong sense of realism, we decided to talk to a published author about the script and also to a real-life "military consultant" about how a soldier of fortune truly lives his life. This was one of the major turning points in the development and we were finally able to focus the game into its final product.

As we settled on an action-movie feel, SoF finally began to take form. We were able to tie together an appealing story line quickly with several twists to keep the player enthralled. Combining this with the extensive amount of information that our military consultant provided us, everyone on the team was excited about the project again and the true development of the game got underway. In less than ten months, the core of SoF was assembled into a fun, viable product. After the game was released this past March, the rest, as they say, is history.

What Went Right

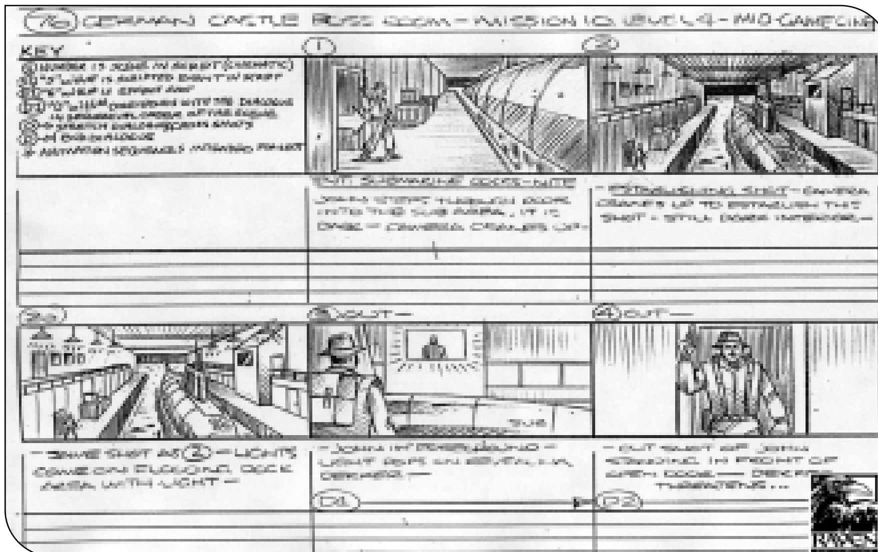
1. Familiarity with technology plus powerful tools and enhancements. One of the most important pluses for SoF was the team's experience and familiarity with the Quake technology. Raven has been using id Software's technology since its early days of Heretic and Hexen. This familiarity allowed us to experiment, create, and use tools that vastly sped up the game's development.

QuakeHelper. One of the first tools that we developed was QuakeHelper. As SoF's development progressed, we realized that all of the options associated with the individual textures for the world were becoming too complex to encode into a parsing file. QuakeHelper was created to allow a visual way to assign all of these properties. This included texture scaling, detail texturing, damage texture (next texture to be shown, and the amount of damage it should take), material properties (sound and visual effects for user interactions), and alternate textures (more detailed and unique textures would be replaced by common textures on video cards with lower texture memory). In the end, SoF had more than 5,000 unique world textures. QuakeHelper saved the artists a tremendous amount of time in preparing the textures for the game and in adjusting and tweaking their properties.

ArghRad. One of the benefits of working in the Quake community is that the public has access to most of the source code to the tools. In the beginning of the project, we used QRAD, which was the original tool id developed to calculate the lighting information on the world. Our designers learned of an enhanced version of QRAD that had been developed by Tim Wright. He called the new modified version ArghRad!, which added a Phong-type shading model to the light map calculation, a global sunlight casting point, and several bug fixes. Raven contacted him to arrange to get the source code. In the end, this helped us create better-looking levels by utilizing the wonderful Quake community.

DS. DS, or Designer Script, was developed jointly for both Heretic 2 and SoF. The goal was to provide a simple language for designers to help create more complex scenes and puzzles in the game. Those who designed this language rightfully kept in mind whom the language was for. In other words, it was a language created by programmers for designers. While this may seem like a straightforward concept, often this idea gets lost during the development phase of tools or other items that are supposed to assist the desired recipient. Even though this language did have certain limitations (described under "What Went Wrong"), it did help meet our goals for both projects. The following two tools helped extend the scripting language in simple yet powerful ways.

ROFF. While one of SoF's designers was playing around with Lightwave to create a



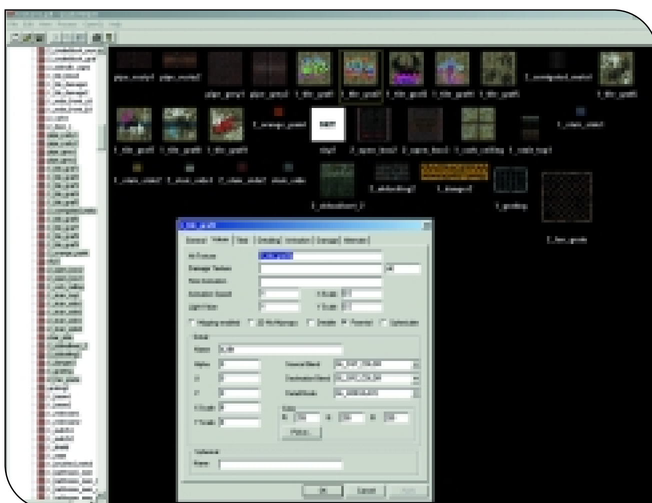
LEFT. Every cinematic sequence was conceptualized with storyboards first.

complex motion path for an entity, he ended up writing an exporter that created a DS script. The script consisted of a series of move and rotate commands to simulate the complex movement animated in Lightwave. While this accomplished the ultimate goal of importing the entity's animation into the game, it was not very efficient. Exporting the movement into a file and adding a command to the scripting language to play that movement file corrected this. This format was known as ROFF (Rotation Object File Format). SoF used about 500 of these movement files, from the simulation of helicopter movement and exploding crates, to even creating a flying bird (although you'll have to look really hard to see that one).

Chimaera. Because of the large amount of animation needed for SoF and the fact that we were going to be using a mix of traditional hand-animated sequences and motion capture sequences, we needed something that would work well with both. All of our motion capture data was taken by House of Moves, a wonderful motion capture house, and sent to our animators. From there, we used Chimaera, a control rig within Softimage that allowed us to tweak both types of animation easily. It also allowed the animators to utilize both inverse and forward kinematics simultaneously, accomplishing this ordinarily complex task with relative ease. One of Chimaera's most important features was that it allowed the animators to apply every animation to any humanoid model, including models not local to SoF. This tool has also been put to good use on our next release, STAR TREK — VOYAGER: ELITE FORCE.

SoFPath. We originally developed SoFPath to create a pathfinding system based on the BSP of a map. During the development of this tool, however, we discovered that the world was broken up too much to provide an effective means of pathfinding. Our early use of .ROFF files also showed that animating entity movement or rotation in a commercial package was difficult without a good representation of the world. Since the SoFPath utility had a good "understanding" of the BSP world, we changed it to export .IFF Lightwave object files. The designers would basically BSP their map (either the full map or a partial region), create the Lightwave file, and import it into Lightwave. They then had a representation of the world, a rough outline of all entities, and could then animate things accurately. Later in the project, we also added the ability to edit these files in 3D Studio Max.

Audio tools. Both dynamic music and ambient sound systems were designed internally to create immersive environments in SoF, but they also allowed the sound designer to add sound assets into the game more easily. Instead of hard-coding the names of the sound files, the tools provided a quick and flexible method of

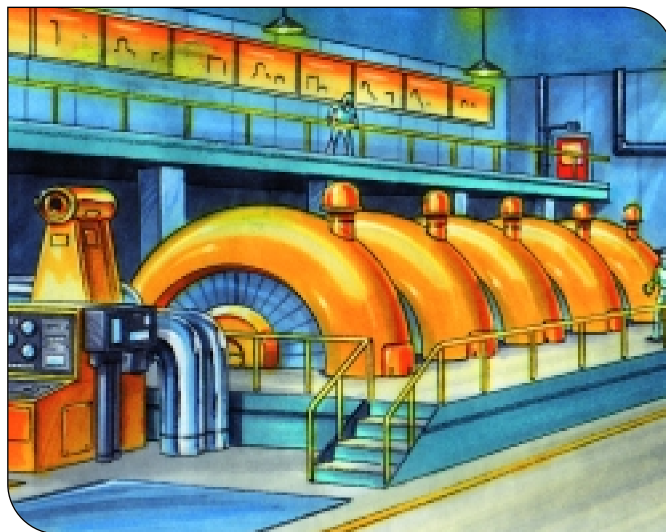


LEFT. Raven developed QuakeHelper to manage more than 5,000 unique world textures with a visual means to assign properties to them.
BELOW. John Mullins, a man for all seasons.

tweaking sonic properties in levels. This process not only took the weight of sound placements off the programmers' shoulders, but also empowered the sound designer with a powerful and creative tool to create unique soundscapes.

2. Taking time to address violence concerns. From its inception, we knew that SOF was going to be a game for adults. Due to its large amount of simulated violence, we wanted to make sure that adults had every opportunity to keep SOF out of the hands of minors while still being able to play the game on their home computers. In order to do this, we implemented several different protective measures for consumers.

First and foremost was creating the SOLDIER OF FORTUNE: TACTICAL NON-VIOLENT VERSION. A totally separate SKU from the regular version of the game, the low-violence option removed all of the gore, limited the number of death animations, and seriously toned down the game in general. This version used the same box as the regular version, but colored red instead of green and stamped with a large advisory that stated that it was different from the regular version.



Off the coast of Russia lies a terrorist chemical weapons plant.

For the regular version, we added a violence-lock feature to allow users to password-protect the game and change various options to their liking. The consumer could lock out dismemberments, blood, death animations, adult textures, and other adult content, essentially turning the regular version into the low-violence version. To further inform consumers of the violent subject matter, a large warning was placed on the front of the box and the ESRB rating was enlarged for greater visibility. A "mature audiences" warning was also added to the game's bumper and implemented into the menu system so that no one would be surprised by the game's content.

All of these features and functions helped extensively in the end. We widened our sales platform as stores realized they could order the tactical version if they wanted to, and we showed consumers that we listened to their needs and concerns, giving them a broader choice in their purchase.

3. Outside help. Although your team will most likely not be using real-life mercenary John Mullins to help design your game, outside individuals can be an incredible help in product development. Talking and working with a person who has an exhaustive knowledge of your game's subject matter will help refine your project and add a truly cohesive feel to the final product. As a consultant helping us with the military aspect of the game, Mullins gave instant feedback in areas where our knowledge was lacking and helped round out the areas that needed it. He described how trained soldiers would react to attacks. He discussed what sounds you would expect to hear in battle. He advised us on how the weapons in the game should "feel" to the player. In short, he helped us to create the correct atmosphere in which to immerse players. By drawing on the insights and knowledge of someone with first-hand experience of the action we were looking for, we were able to focus the design of the game.

4. "Commando" marketing and buzz words. We knew that in order to keep the QUAKE 2 engine competitive in the FPS realm, we had to add significant technology. Many of the technology improvements we made were centered on new modeling technology, which featured, among other things, a completely new modeling system, compression of animation data, attachment of models (bolt-ons), multiple skin pages per model, and advanced networking.

Our lead technology programmer dubbed this new modeling system Ghoul (in keeping with an earlier in-house technology proposal called Specter). In the public's eye, we associated all of these major changes with the Ghoul name. Without Ghoul, SOF would have been a mere shadow of the final product. It allowed us to throw in all the bells and whistles, including the vast array of enemies and the high degree of gore. As SOF's development progressed, our continued references to Ghoul caused the public to monitor the changes and build up their expectations. Ghoul became an important marketing word for SOF.

Besides normal marketing channels such as magazines and print ads, we decided to try our hand at "commando" marketing. By using our .plan files, giving web interviews, supporting the wonderful fan sites that were popping up, and making ourselves available through e-mail and online chats, we established a strong presence in the Internet community. This proved invaluable for consumer feedback. With the release of the demo and the early OEMs, players gave us instant feedback on what they liked and disliked and we were able to change the game accordingly. One example where this feedback came in handy was with limited saves. Originally, players were limited in the number of saves that they could make based on their present difficulty level. Many people who played the demo disliked this feature, so we added the ability to customize the number of saves that players could make, thus adapting the game directly to consumers' preferences.

5. Good planning and scheduling. One of SOF's saving graces was that it was planned and scheduled well. The sheer volume of animation, art, programming, and levels forced us to update our schedules on a frequent basis.

ABOVE. Color, mood, and scale were all considered in concept art, not just the architecture. RIGHT. Breaking out the big guns. FAR RIGHT. Enemies were created to reflect the locations in which they would be found.

With a concrete animation naming system, an incredibly large and detailed database for animations, storyboards for every cinematic sequence, and a well-designed QA system, SoF did not suffer much inefficiency. The only area that endured some wasted time was the design due to the various story and game changes.

Once the story was finalized and had the green light, establishing and maintaining good planning and scheduling for the design process helped finish the game in a timely manner. We created total level walkthroughs, with each room and encounter written out. Flowcharts were used to draw the preliminary levels, and concept art was used for key location elements. Perhaps the most important lesson we learned from SoF was that preplanning is the most important aspect of game creation.

What Went Wrong

1. Unfocused design. The single most damaging problem during SoF's early development was that the original game lacked a truly focused design. We knew what the fundamentals of the game would be, but we did not have the specifics that we needed to create a solid, cohesive product. The game's overall story changed five times before it was finalized — at one point we had even changed the basic game concept to a team-based tactical shooter, similar to RAINBOW SIX.

One reason for this indecisiveness was that, at the time, our original marketing team was wondering what the "hook" would be for the game. This was a major roadblock in creating the game because we knew that if marketing wasn't behind the idea, SoF wouldn't get the marketing money that it deserved. On top of that, without the backing of the marketing division, the senior management at Activision wouldn't get behind the title, either. We had to constantly sell and resell the idea that a high-octane, action-movie-like, real-world combat game would be enough of a hook. At times, it went so far that we were making design decisions not for the fun or betterment of the game, but to find the hook that we felt we were missing. The last straw came when we found ourselves working on a tactical team-based shooter, a complete 180-degree shift from our original design. We then decided to return to the game's roots

and started banging out a new story.

Eventually, a new marketing team came on board that recognized exactly what we had been saying all along. SoF had more than enough to stand on its own, and they worked with us to find the right angle for marketing the product. This new team fit right in with the development team and things started to roll. On top of that, since we urgently needed to nail a story down in a short amount of time, they recommended that we meet with a hot-selling writer (Gonzalo Lira, author of the spy novel *Counterparts*) and John Mullins. Although the full story that Gonzalo Lira wrote for us was never used, some elements of it were, and the process made us realize exactly what we wanted from this game and how to get it. John Mullins contributed an element of realism to the game that we were missing at that time.

In short, working on everything at once was not the way to go. For the projects that we currently have lined up we are designing the entire game from start to finish before we begin physically developing it. The SoF team learned the hard way that a day of preplanning saves a week of rework. Also, getting a green light for everything before starting development saves having to back-pedal later on. Both of these lessons will be applied to our future projects.

2. Technology creation took longer than expected to visualize gameplay. A sure way to sell your product is to have a working prototype at an early stage in its development. Since we had decided to give the Quake 2 engine an entire overhaul, we realized that we would really have to come together and work as a team to make sure things were completed on time.

One of the major enhancements for SoF was the Ghoul modeling system, which replaced the entire Quake 2 modeling system, and turned out to be quite the undertaking. Throughout the entire life of the project, tweaks and changes were made to Ghoul to make it more flexible and powerful. Unfortunately, this also meant that for a substantial part of the early development, we had no game to look at — only individual components. It's one thing to be able to look at a model in a model viewer, or at a level with nothing in it, but it's essential to be able to see the model in the world and interact with it.

Another problem was the huge number of animations planned for SoF. Since we had so many animations (more than 600 sequences) we had to limit which animations would appear on a specific level due to memory constraints. Limiting the animations on a per-level basis was a nightmare in itself, not only for the animators but also for the AI programmer (who had to make the AI work within the animation constraints) and the designers, who had to create scripted and cinematic sequences using only the animations available for each level. As the game drew nearer and nearer to completion it became increasingly difficult to bring new animations into the game without ruining someone else's work by removing an animation that was already in use.

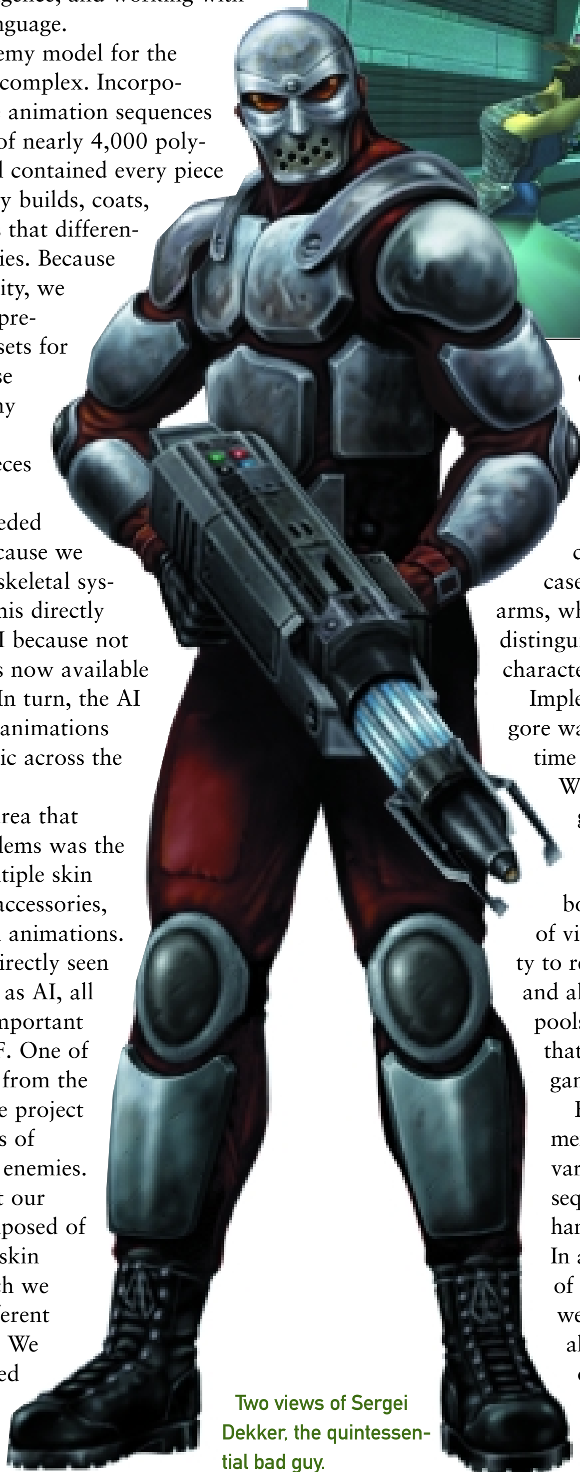
The final problematic technology was the AI. Developed throughout the entire course of the project, the AI went through many different incarnations. We decided early on that the pace of the game should

BELOW LEFT In-game cinematic sequences helped establish the action-movie feel for the game.
BELOW RIGHT. Cleaning up the New York subway system.

be fast and furious with a large number of enemies attacking at once. Enemies were to be reactive, but not too intelligent. There were three major areas that caused AI problems: developing the models, developing the intelligence, and working with the scripting language.

The main enemy model for the game was very complex. Incorporating all of the animation sequences and consisting of nearly 4,000 polygons, the model contained every piece for various body builds, coats, and other items that differentiated the enemies. Because of this complexity, we were forced to preprocess enemy sets for each level. These individual enemy sets looked at what enemy pieces and animation frames were needed for the level because we did not have a skeletal system in place. This directly impacted the AI because not every move was now available on every level. In turn, the AI could only call animations that were generic across the levels.

The second area that caused AI problems was the addition of multiple skin pages, bolt-on accessories, gore, and death animations. Although not directly seen by most people as AI, all of these were important features for SoF. One of our main goals from the beginning of the project was to have lots of unique-looking enemies. This meant that our model was composed of many different skin pages into which we could swap different faces or outfits. We also implemented what we termed "bolt-ons": any item or feature that was not originally part of the model. These included Mohawks, canteens, briefcases, and side arms, which helped distinguish different characters.



Two views of Sergei Dekker, the quintessential bad guy.

Implementing the gore was also very time consuming. We implemented gore zones that required skin page overlays, bolt-on models of viscera, the ability to remove limbs, and all of the blood pools and spatters that litter the game.

Finally, implementing the various death sequences also hampered the AI. In addition to all of the gore that we created, we also had to play one of several animations when an enemy died. Animations had to be called based on certain circumstances, such as where on the body the enemy was shot and what he was shot with. On top of all of that, adding the violence-lock system that would allow players to lock out the game violence meant that all of the gore and animations had to be able to be shut off if the player wanted.

The third area that caused problems for the AI was its actual development. Along with the problems created by the per-level animation system, the AI also had to work with the game's scripting language. If the AI was tweaked in certain ways, it caused the scripting to break. Many times in the game, enemies had to be "frozen" in place while their script waited to be activated. If a player happened to see one of these suspended enemies before they were triggered, it obviously made the AI appear less intelligent. We had to come up with ways around these problems, and expended considerable time and energy to fix them. To make matters worse, the AI had a slight unpredictability built into it that caused scripted events to occur differently each time. Although unpredictability is good for gameplay, it had to be removed from the scripting element. Finally, a large amount of time was spent with the designers to build in hints for the areas (such as reactions of the enemies) and to specify which areas the enemies could traverse. At the beginning of the development we had "duck," "hide," "flee," and other commands that eventually were removed and taken over totally by the AI. The AI was in development until nearly the end of the project.

3. Too many OEMs and demos. Something that seemed like a great idea at the time but turned out to hurt us in the end was the decision to make specific OEM releases before the game was truly finished. The main reason for this was that we looked at the revenue that would help the bottom line instead of considering how much it would set back the game.

Because there were both regular and low-violence versions of the game, we needed to make several different builds for the different violence levels and test each build accordingly. In the end, we had roughly 75 QA submissions. While each OEM and demo iteration helped bring more of the game together, it also diverted our attention from the final product. As we were tweaking and fixing the OEM versions, full production would come to a standstill as we focused on getting the smaller versions out the door.

4. No fixed deadlines. Originally, SoF was scheduled to ship in July 1999. Activision wanted to avoid releasing SoF in the "blast zone" of competing FPS titles that were shipping that year, so they extended the deadlines on the game. As our competitors' titles were pushed back, so was SoF. Although within these deadlines we had schedules set up and planned out, this caused a never-ending uncertainty of how much time we had left in the project and how much technology we could add or change within that time.

In March 1999, we realized that with our complex models and the amount of animation we wanted, we needed to address memory concerns. Because we thought that we only had three or four months left of core development at that stage, we concluded that switching to a skeletal system would be too risky for the project. Instead, we created a vertex compression system that mimicked the benefits of skeletal compression in a few ways. Unfortunately, this meant that we were not able to provide all of the animations at once, as we would still be over memory budgets. If we had known that our deadlines would be pushed back another six months, we would have added the skeletal system, saving everyone a large amount of headaches and work.

5. Miscommunication about some technologies. Confusion over project scheduling aside, additional technologies were developed during the course of the project that were never truly planned out appropriately, such as the terrain engine, the in-game effects editor, and the scripting system that we used. All of these technologies served to improve the game substantially, yet they could have worked better if they had been properly discussed between the team members.

The terrain engine, while flexible enough to do various types of visual effects, was never properly coded into the gaming logic. The basic premise of the terrain engine was that the designers would create architecture that represented the portions of the world that the player could interact with. For example, on the train level, the train was created by the designers. The terrain engine would then be responsible for the scrolling polygons, in this case the train tracks and surrounding landscape. When we put this level in, we soon realized that we needed a bunch of special code to handle the various effects, such as when a person falls off a train. We wanted to add more unique kinds of levels like this but we didn't have time to develop a generic physics system for handling other types of terrain, such as water where bodies might float or sink.

The effects editor was created by one of the programmers to help him create visuals for the weapons. The interface, while functional, was crude. Other people wanted to create visuals, including designers, but were hampered by the editor's interface design since it was never intended to go beyond the programmer who created it. Although the in-game editor allowed someone who knew the tool to create a special effect quickly and efficiently, it had a long learning curve for those not familiar with it. This reduced the amount of control that the artists had over the effects.

SoF shared the same scripting language that HERETIC 2 had used. It was originally developed to give designers more control over their levels, but we soon learned that we would need to add more and more power to the scripting system. SoF's complex scripting soon overwhelmed the scripting language and too much time was spent trying to tweak out sequences. With the addition of in-game cinematics (an unplanned feature not included in the design document), we realized that the way we were using the powerful scripting language was wrong. If we had planned better from the beginning, the scripting would have gone much easier. Unfortunately, since the story was planned so late, we didn't know at the time what would be needed.

This character appears unperturbed by deadline uncertainties.

A Direct Hit

Originally slated for an 18-month development cycle, SOLDIER OF FORTUNE ended up taking nearly two years, a considerable undertaking that in the end allowed a talented group of developers to really shine. As with all projects, SoF had its problems, but for the most part things went well thanks to the efforts of an incredible team of people, and SOLDIER OF FORTUNE has quickly become one of Raven Software's best-accepted titles. With strong sales to date and a solid Internet community, SoF has exceeded many people's expectations, including our own. We've been very happy for the large number of good reviews, both in print magazines and on the Internet, and we are helping to support the online community as much as we can. From a development viewpoint, SoF allowed the Raven team to grow and mature, and many lessons that we learned are now being put to use in our next set of products. Of course, no project ever runs smoothly, but with each new game we gain more understanding of what it takes to make the next one better.

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

Pay No Attention to the Orchestra Behind the Curtain!

So, the debate about interactive music rages on... oh, wait — there's no debate! Instead every audio department in games is scrambling to use or develop new tools for interactive music, the great panacea. Developers are thrilled with the idea that by implementing a soundtrack that changes over time according to the behavior of the user we'll be able to provide the ideal (yet ever-elusive) immersive experience. We've finally got the solution to our musical problems in our sights: at last, the music can be as interactive as the rest of the game. While this is all very exciting, before we throw away everything we know about music and invent a new musical language (and it would take nothing less to make this work), let's remember what we're really trying to accomplish.

How interactive is "interactive music," anyway? Frankly, not much. Let's address some semantic issues first. What we're after is not actually "interactive" music, but "adaptive" music. Interactive music is music that one interacts with in order to change it. The result is the music itself. In our case, the player interacts with the game, and the result is the game. All other results — graphical, auditory, textual, or otherwise — are ancillary. The gameplay is what matters. So what we are looking for is music that can *adapt* itself to always be appropriate and compelling based on the current state of the game, with the goal of enhancing the real product, gameplay. In fact, the last thing we want is for the player to interact with the music — God forbid we put in a trigger to intensify the music and the player finds it and plays with it, walking back and forth turning the sustained strings or the pounding timpani on and off instead of playing the game. No, we definitely do not want interactive music.

But adaptive music in a game environment has its own set of difficulties. A good soundtrack pushes and pulls the listener along, hinting at things to come, building to false climaxes just before real ones, sometimes adding depth by (briefly but intentionally) sounding inappropriate, and so on. These are hard tricks to perform in a game environment. Mechanisms have to be built that manipulate the music in a manner that feels totally natural, yet are not so obvious as to be readily apparent to the player (or we're back to the "interactive" problem I just described). Since an interactive environment lacks the prescience of a linear story, we can't hint at what's about to happen. So we must either dispense with this technique and just have the music react to the game, or allow the music to foreshadow generically in order to cover all possibilities (leading, potentially, to great banality in our music). What we end up with is music that is called "interactive," but is actually trying to be "adaptive," and only really manages to be "reactive" — and is boring to boot.

Is "reactive" music better than linear music? It depends on the context; it certainly could be. But in a quest to surmount all these obstacles to implementing adaptive music, how much are we willing to

continued on page 71

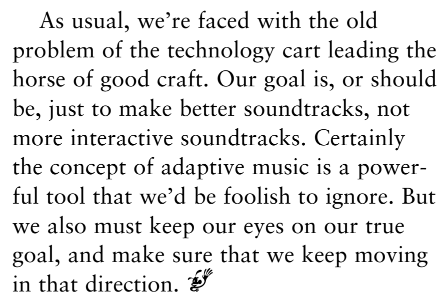
illustration by Dominic Bugatto

continued from page 72

sacrifice of what we already know about making good music — about music's internal logic and flow, its affective power, its mysterious ability to communicate without concrete symbols? Is a truly adaptive soundtrack of bland and uninteresting music a reasonable goal? And even if we achieved an adaptive soundtrack that didn't consist of bland music, would we be contributing to or detracting from gameplay? Imagine the following scene in a movie: after passing through a series of rooms strewn with items in various states of disarray, our hero walks down a darkened hall, toward a mysterious door. As he approaches it the music swells discordantly and the audience leans forward tensely, now certain that horrible evil awaits him on the other side of the door. Done right, this kind of scene can be very effective. And we can finally do it in games, too!

But if we examine why this scene so musically effective, it doesn't seem to map onto the game playing experience. In the movie, if the audience identifies with or is at all invested in the character, they don't want anything bad to happen to him. And the music is communicating that something very bad is about to happen — thus the tension. But here's the crucial difference: the music is communicating this to the audience and *not* to the character. The moment is scary and tense because we (as the audience) know something that the character does not. "Please don't keep walking down the hall, can't you hear the scary music? Go back to the previous room and pick up that fire poker you saw lying there!" we want to shout at the screen. Except in certain instances of so-called diegetic music (someone clicks on a radio, tickles a piano's keys, a band plays in the background), it is always the case in movies that the soundtrack is for the audience exclusively and not for the characters. But this can't be true for games — if the player is controlling the character and driving the action, then the player is both audience and character.

So the fact that in a game the character "hears" the music (in a sense) changes the music's function. In the scene I just described, imagine if we were to believe that the character could hear the music — the scene would unfold as comedy or we would lose all faith and interest in the character. The same is true of a game. If I'm playing a game and I'm about to have my character open a strange door and the music suddenly gets ominous and portentous, why would I then open the door? Instead I think, "Hmm, it seems like something bad is going to happen, maybe I should go back and get that fire poker lying in the previous room." This is probably not the intended effect — instead of creating a more immersive environment, the music has just pulled me out the fiction entirely. Might as well have had a voice-over say, "Don't open the door yet — go get the fire poker." Of course, it could very well be that this is indeed the gameplay sequence the designer sought, in which case the music has become more "interface" than "soundtrack." Instead of making me (the player as audience) feel scared, it gave me (the player as character) a clue about how to play the game — and a clue not intrinsic to the game world. Is that the best use of a soundtrack?



As usual, we're faced with the old problem of the technology cart leading the horse of good craft. Our goal is, or should be, just to make better soundtracks, not more interactive soundtracks. Certainly the concept of adaptive music is a powerful tool that we'd be foolish to ignore. But we also must keep our eyes on our true goal, and make sure that we keep moving in that direction.

ANDREW BOYD | *Andrew is sound design manager at Stormfront Studios. Contact him at aboyd@stormfront.com. He's not really this curmudgeonly. Honest.*

ADVERTISER INDEX

COMPANY NAME	PAGE
Apple Computer	2
Ascension Technology	59
Compaq Professional	31
Conitec	27
Criterion Software	7, 51
Cyan Inc.	68
Daily Radar.com	61
Dice.com	68
Havok.com	5
HotGen Studios	69
Improv Technologies	22
Inoiz.com	47
Macrovision	C3
Metrowerks	28
Microsoft Developer Network	11
Morgan Kaufmann	8
Motek	19
Newtek	21
Numerical Design Ltd.	14
NxN Software AG	C2
Pulse Entertainment	33
RAD Game Tools	C4
Rainbow Studios	68
S3 Inc.	17
Savannah College of Art	70
Seneca College	70
Sony Computer	67
Staccato Systems	35
Vancouver Film School	70