



GAME DEVELOPER MAGAZINE

AUGUST 2003





GAME PLAN

LETTER FROM THE EDITOR

Life in the Majors

“...A collection of 25 youngish men who have made the major leagues and discovered that in spite of it, life remains distressingly short of ideal.”

So renowned baseball author Roger Kahn once defined a major league baseball team, but I couldn't help thinking about how equally well he could have been describing a professional game development team. In particular the number 25, rather arbitrary in Kahn's original context, caught my attention. Why has this number remained game development's project management glass ceiling for the past few years?

It's trite to describe any particular moment in the history of game development as “transitional”; the dynamic nature of the technology allows a nascent form of creativity to pour into an ever-expanding, seemingly insatiable vessel. Growing from teams of eight to 15 that characterized the previous generation to teams of 20 to 25 that characterize today's did not prove as chaotic as originally feared. Loose, informal organizational structures, planning, and feedback stretched the project management spiderweb, but it did not snap.

Now, in the middle of the current hardware cycle, is an unusual time to see a cleft widening between the under 25-to-30-person teams and those that are becoming larger. This magic number 25 emerges as a boundary between the viability of ad hoc, heuristic decision-making that characterized the first 25 years of game development, and the need for more deliberate and concerted software risk evaluation and management.

This tension has left many small, independent game developers straddling an uncomfortable fence. Many smaller developers lead a perilously hand-to-mouth existence, making organic growth difficult to impossible. To add a second project team en masse is to require a completely new (and hence locally untested) methodology for all involved.

In game development particularly,

experience acts as both ally and enemy when experimenting with new software management schemes. Processes may be outmoded or inherently flawed, yet the sensibility of throwing them out the window is betrayed by the strange comfort of old habits. So I became interested when I caught wind of how San Diego-area Sammy Studios was trying to manage their technological development differently (in the sense of “correctly”) from day one, which is the crux of this month's cover feature by Sammy's Clinton Keith (beginning on page 28).

I was already keenly aware of the quiet deftness with which Sammy had been attracting a rich array of talented names across technology, design, and visual development, and I wanted to find out what happens when a group of successful veterans comes together to try to build a better mousetrap. Talk is cheap, and for those who could benefit from a rigorous reengineering of their own development management, I hope the decision of Sammy Studios' parent Sammy Corp. to invest substantially in a centralized technology group will serve as a model for other multi-project companies. They've rationalized the relationship between consumer expectations and development risks in order to maximize the former and minimize the latter.

Rather than being strictly tied to hardware, it is in balancing this equation favorably that the next wave of game development's growing pains lies. The ability to amortize risk across several projects with larger teams and core assets is already heralding a new age of haves and have nots, surprisingly in advance of the next hardware cycle. How well the small-team development management approach will survive remains to be seen; right now, applying Kahn's observation, it seems it may fall farther short of ideal with ever-increasing distress.

Jennifer Olsen
Editor-In-Chief

GameDeveloper

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

Publisher
Jennifer Pahlka jpahlka@cmp.com

EDITORIAL
Editor-In-Chief
Jennifer Olsen jolsen@cmp.com

Managing Editor
Everard Strong estrong@cmp.com

Product Review Editor
Peter Sheerin psheerin@cmp.com

Art Director
Audrey Welch auwelch@cmp.com

Editor-At-Large
Chris Hecker checker@d6.com

Contributing Editors
Jonathan Blow jon@number-none.com
Hayden Duvall haydend@3drealms.com
Noah Falstein noah@theinspiracy.com

Advisory Board
Hal Barwood LucasArts
Ellen Guon Beeman Monolith
Andy Gavin Naughty Dog
Joby Otero Luxoflux
Dave Pottinger Ensemble Studios
George Sanger Big Fat Inc.
Harvey Smith Ion Storm
Paul Steed Microsoft

ADVERTISING SALES
Director of Sales/Associate Publisher
Michele Sweeney msweeney@cmp.com t: 415.947.6217

Senior Account Manager, Eastern Region & Europe
Afton Thatcher athatcher@cmp.com t: 828.350.9392

Account Manager, Northern California & Southeast
Susan Kirby skirby@cmp.com t: 415.947.6226


Account Manager, Recruitment
Raelene Maiben rmaiben@cmp.com t: 415.947.6225

Account Manager, Western Region & Asia
Craig Perreault cperreault@cmp.com t: 415.947.6223

Account Representative
Aaron Murawski amurawski@cmp.com t: 415.947.6227

ADVERTISING PRODUCTION
Advertising Production Coordinator Kevin Chanel
Reprints Terry Wilmot t: 516.562.7081

GAMA NETWORK MARKETING
Senior MarCom Manager Jennifer McLean
Marketing Coordinator Scott Lyon

CIRCULATION

Group Circulation Director Catherine Flynn
Circulation Manager Ron Escobar
Circulation Assistant Ian Hay
Newsstand Analyst Pam Santoro

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

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

CMP MEDIA MANAGEMENT
President & CEO Gary Marshall
Executive Vice President & CFO John Day
Executive Vice President & COO Steve Weitzner
Executive Vice President, Corporate Sales & Marketing Jeff Patterson
Chief Information Officer Mike Mikos
President, Technology Solutions Robert Falettra
President, Healthcare Media Vicki Masseria
Senior Vice President, HR & Communications Leah Landro
Vice President & General Counsel Sandra Grayson
Vice President, Applied Technologies Philip Chapnick
Vice President, Electronics Paul Miller
Vice President, Software Development Peter Westerman
Vice President, Information Technologies Michael Friedenber
Corporate Director, Audience Development Shannon Aronson
Corporate Director, Audience Development Michael Zane



GamaNetwork



INDUSTRY WATCH

KEEPING AN EYE ON THE GAME BIZ | *everard strong*

Double-digit growth for games. PricewaterhouseCoopers' annual "Entertainment and Media Outlook" report predicts that global spending on the entertainment and media industry will surpass \$1.1 trillion in 2003, rising by 3.7 percent from its 2002 level. In particular, the firm forecasts that growth for television distribution, videogames, Internet access, and home video will be spurred by a 30 percent compound annual growth rate (CAGR) in broadband. This could translate into more than 153 million broadband-enabled homes worldwide by 2007. The report also forecasts double-digit CAGR increases for videogames between 2003 and 2007, and singled out the game industry as the fastest growing entertainment/media segment, outpacing Internet advertising and access spending. The report identifies two areas that will drive spending; online videogames and mobile phone games. Growing at an 11 percent compound annual rate, next-generation consoles will drive the industry's worldwide market to \$35.8 billion in 2007, the report predicted.



Industry veterans from DMA and Psygnosis, creators of classics such as LEMMINGS, have joined to found new studio Real Time Worlds.

DMA, Psygnosis founders launch new studio. Three game industry veterans in the U.K. have officially unveiled their new development studio in Dundee, Scotland, called Real Time Worlds. The three launching the 28-person studio consists of David Jones, who founded DMA Design in the 1980s, the company that launched the LEMMINGS and GRAND THEFT AUTO franchises; Ian Hetherington, former managing director of Sony Computer Entertainment Europe and founder of Psygnosis; and Tony Harman, the former director of development and acquisition at Nintendo of America and the

executive producer of DONKEY KONG COUNTRY franchise for Nintendo.

Over 1 million sold. Nintendo announced that their Game Boy Advance SP has sold over 1.1 million units since its May 2003 introduction.

3DO bankrupt, delisted from Nasdaq. 3DO filed for Chapter 11 bankruptcy protection in the United States Bankruptcy Court for the Northern District of California, and will sell off the company and/or its assets. In a public statement, CEO Trip Hawkins said that the company is expected to continue to operate as it works through the bankruptcy process. Hawkins was keeping the company afloat with his own funds as company cash reserves grew low. Nasdaq had delisted 3DO's common stock as of June 9.



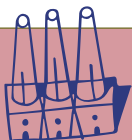
Send all industry and product release news to news@gdmag.com.

UPCOMING EVENTS CALENDAR

EDINBURGH INTERNATIONAL GAMES FESTIVAL
EDINBURGH INTERNATIONAL CONFERENCE CENTRE
Edinburgh, U.K.
August 18–19, 2003
Cost: £89 (+VAT)
www.eigf.co.uk

GAME DEVELOPERS CONFERENCE EUROPE/ECTS
EARL'S COURT CONFERENCE CENTRE
London, U.K.
August 26–29, 2003
Cost: £89–£388 (+VAT)
www.gdc-europe.com
www.ects.com

AI DEVELOPMENT WORKSHOP
UNIVERSITY OF TEXAS AT AUSTIN
Austin, Tex.
August 21–23, 2003
Cost: \$95–\$445
dmc.ic2.org



THE TOOLBOX

DEVELOPMENT SOFTWARE, HARDWARE, AND OTHER STUFF

Discreet Particle Flow tools released. Discreet announced the availability of its new Particle Flow extension software for 3DS Max customers. The Particle Flow extension integrates into 3DS Max for the creation of effects such as fountains, fog, snow, splashes, contrails, explosions, and other environmental effects. www.discreet.com

Synergenix announces Mophon 3D. Synergenix has launched the Mophon 3D engine, featuring low memory footprint and low processing power requirements for its suite of 3D APIs. The engine is designed for development of 3D mobile games that are

compact and downloadable over-the-air. Also included are features such as enhanced audio, multiplayer, and more. The engine still maintains full binary backward compatibility with all currently existing Mophon games. www.synergenix.se

Singular Inversions unveils FaceGen Modeller 3.0. Singular Inversions has released the latest version of its 3D face-generator software, which includes such new features as click-and-drag free-form deformations, a genetic face-creation interface, more file export options, and user interface improvements. www.facegen.com



AI Middleware: Getting Into Character

by eric dybsand

After 16 years of designing and developing artificial intelligence solutions for many different genres of computer games, I was admittedly skeptical that any AI middleware product could help the computer game AI development process in any significant way. After all, I thought, the AI for every game is game-specific, performance-sensitive, and thus requires an experienced computer game AI craftsman such as me to design and develop good AI solutions. Then I began researching the various AI middleware products for this review.

Could I be wrong? Computer game AI middleware is software that provides services to game engines for performing the AI function in computer games. Computer game AI middleware will often find itself outside the game engine and the process of producing the actual behavior of agents or non-player characters (NPCs) or decision-making objects found in a computer game (Figure 1).

Often developed by companies that specialize in AI solutions, AI middleware has reached a noteworthy level of maturity.

This review will summarize four character-oriented AI middleware products, all of which are being used in game development projects as of this writing. You will also find advice on how an

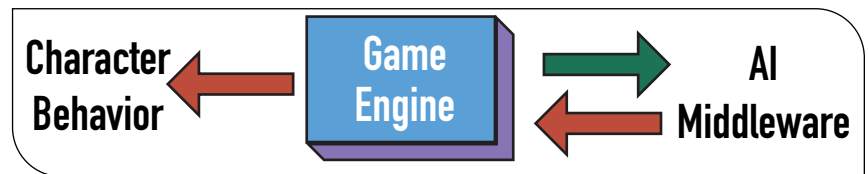


FIGURE 1. Typically, character and game state status flow from the game engine to the AI middleware, and then character control requests flow from the AI middleware to the game engine, and are acted out by the characters.

interested game developer should evaluate AI middleware. Since this review provides only a summary of the features of these products, I encourage the interested reader to read the expanded, more detailed reviews of these products that can be found online at *Game Developer's* sister publication, Gamasutra.com.

Why use AI middleware? One reason game developers may choose to use AI middleware in their games is that the internal staff may not possess the AI expertise to develop the desired AI algorithms and processes for the game. Or perhaps there is a specific (or rare) AI technology that the game designer desires to be a part of the game's AI. Another reason might be that the remaining game development time available before shipping may be insufficient to internally develop the desired level of AI for the game, whereas an AI middleware product selected contains the algorithms or processes that may achieve the level of AI desired by the game developer.

Why not use AI middleware? The most common reason for not using AI middleware is probably the issue of the “not invented here” syndrome and the game developer's fear of not having complete control over all game processes. Also of concern is a perceived performance hit that may result from having to rely on the AI middleware “engine” or library routines for some processing but then not being able to optimize the code to the performance level desired by the game developer or AI programmer. The learning curve of an AI middleware product may also prove to be too significant. And there is the possibility that the AI middleware may not do exactly what the game developer wants to have done, hence forcing modification of externally developed software at a critical time in the game development process.

AI Implant: An Animation Control AI

AI Implant, the sophisticated animation control engine developed by Biographic Technologies, introduces AI to the character development process. The AI Implant production pipeline fea-

ERIC DYBSAND | Eric has been involved with computer game AI since 1987, doing game design, programming, and testing for a variety of genres of computer games. He has been a speaker on computer game AI at the GDC for the last seven years and is a contributing author on AI to the *Game Programming Gems* and *AI Wisdom* series of books.

tures Maya and 3DS Max plug-ins that allow AI to be created for a character after the modeling process. Once a character has been created in Maya or Max, the AI Implant plug-in can be used to set animation control for the character, add behavior-related attributes to the character, set default and initial state values, add sensors to the character, assign behaviors to the character, and create decision trees to be assigned to the character to manage its behavior.

Characters can be grouped and coordinated using AI Implant. Default behaviors can be assigned to characters to execute in lieu of complex decision making. Decisions are achieved using decision trees, which can be assigned to characters via the plug-in interface, allowing for rather complex and rule-based decision processing. These can also be used as finite state machines (FSMs) for state-driven AI. AI Implant also offers waypoint editing (through the plug-ins) and automatic waypoint network generation for the virtual game world (using existing geometry/physics markup) that can be used by the runtime pathfinding algorithm.

The most exciting aspect of AI Implant for me is the Maya and 3DS Max plug-ins. Creating the AI was basically “create and play,” so experimenting with different behaviors was easy. However, I felt constrained in the types of decision-making techniques I could use within AI Implant, which seemed limited to the decision trees.

AI Implant is unique among AI middleware products and could be useful to a number of game developers, especially those using Maya or 3DS Max for modeling and animation who wish to offload some of the AI development to a designer working with the art department.

DirectIA: An Adaptive Behavior SDK

DirectIA (Direct Intelligent Adaptation) is a generic SDK for game AI developed by the MASA Group. It relies on several built-in engines for processing. There is a motivation engine to

AI Middleware Products	AI Implant	DirectIA	Renderware AI	Simbionic
User Decision Support	Decision trees, finite state machines	Motivated decision graphs	Finite state machines, neural networks	Hierarchical polymorphic finite state machines
Other Services	Automatic waypoint network generation, pathfinding, AI animation control, LOD	Pathfinding	Graphics, physics, auto path generation, pathfinding	Built-in communication between agents
Behavior Support	Prepackaged behaviors	Template behavior scripts	Prepackaged behaviors	User-developed behaviors
Engine Source Code Availability	Some	No	Yes	With license
Extensibility	User-developed behaviors, sensors, space partitions, world and object geometry	User-developed scripts, callback functions	User-developed behaviors, callback functions, Lua, brain-scripting	User-developed behaviors, callback functions
Production Tools	Maya/3DS Max plug-ins	Script templates, tuning GUI	AI skeleton code, XML configuration	Visual Editor, Visual Debugger

TABLE 1. Summary of common features of reviewed AI middleware products.

model the emotions and needs of the agents, a behavior engine to model the agents’ decision processes, an action engine to enable the agent to interact with the game world, and a knowledge engine to organize the agent’s understanding of the game world.

DirectIA offers real-time decision and action behavior modeling tools. Within the high-level tools there is support for complex agents and reactive agents that is driven by the engines mentioned above. The low-level tools offer pathbuilding, hierarchical pathfinding, and steering tools (in beta at this time).

As a tool, DirectIA is very agent-centric. What impressed me the most about DirectIA was the scope of its sophisticated behavior engine. However, using the DirectIA behavior engine was restrictive; the engine relied on its own built-in functionality tuned via script files, instead of my coded alternatives, to meet the agent decision-making needs of the game developer. Communication and perception for the agents can be integrated via user-defined callback routines accessed from the scripts. The decision-making processes of DirectIA suggest that it is also very state-oriented. Since many games rely on FSMs to help make decisions for characters, DirectIA could fit right into most genres.

Renderware AI: An Architecture of C++ Classes

Renderware AI (RWAI), developed by Kynogon, a French company specializing in game AI (the Kynogon Artificial Intelligence Modules, or KAIM), is packaged as part of Criterion Software’s Renderware Platform suite of game development products and tools. Discussion about the entire suite of Renderware products, which also includes graphics, audio, and physics tools, is beyond the scope of this review.

The RWAI SDK primarily focuses on helping the game developer design and implement character behavior in a game. RWAI views the objects in the world as entities in two basic forms: thinking entities (NPCs) which have brain objects, and passive entities (objects that exist in the world and interact with thinking entities). RWAI provides several layers of services:

- Decisions support the brain objects of thinking entities.
- Agents support behavior carried out by entities.
- Services provide specialized manager objects.
- Architecture provides interface to the game and configuration services.

The entire Renderware suite of tools

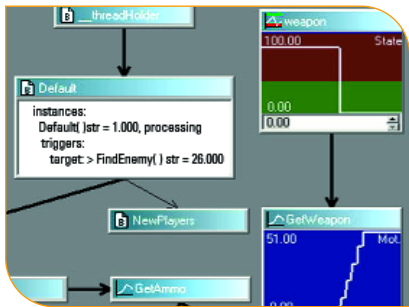


FIGURE 2. Screenshot from DirectIA's interface.

offers a game developer a complete development environment, thus the compatibility of these components places RWAI in a unique situation among other AI middleware offerings. It is the KAIM, which provides the core AI power to RWAI yet constrains the user's options a bit for my taste, that caught my attention. Despite the minor limitation, RWAI offers a powerful AI middleware SDK to the game developer, and the use of RWAI in combination with the other Renderware Platform components provides a complete solution to game developers. Even in a stand-alone configuration without the other Renderware Platform components, RWAI is a sophisticated solution that is worth examining in detail.

Simbionic: Authoring Tool for Complex Behaviors

Developed by Stottler Henke, Simbionic is a visually oriented AI middleware product for character definition and control, decision making, and behavior assignment. It is also very state-oriented; the control flow is influenced by the state or condition of some object or process, the same way finite state machines operate. The Simbionic state systems have many components that can be classified as descriptors and declarations:

- Descriptors help to describe objects and attributes.
- Declarations create symbolic associations to use in definitions.
- Entities define NPCs, agents, and objects in the world.

- Actions declare behaviors that entities can perform.
- Behaviors dynamically determine decisions and actions performed by entities.
- Predicates provide built-in and user-defined access functions.

Simbionic provides a sophisticated framework for creating and debugging state systems. The Visual Editor makes the development of these state systems easy and “designer-accessible,” because after the initial construction of the elements, little additional coding is involved in order to assemble the AI. This visual-editing feature was my favorite user interface out of all the AI middleware products evaluated. Dragging and dropping behaviors seemed very easy for any user to grasp. That drag-and-drop functionality relies on the development of user-defined actions, and predicates as code modules, which means that the ease of use for the visual editing must also be supported by the programmer developing appropriate code.

A Simbionic user I contacted was amazed that once he started working with the program, it was so easy to create and modify the AI portion of his program. With Simbionic being so state-centric and FSMs so widely used by game developers, Simbionic products could be an alternative to the custom-code FSM development that goes on in game development today.

Wrap Up

In his article “Effective Middleware Evaluation” (May 2003), Alex Macris made many valid points about the process of evaluating middleware products for game development. I concur with his conclusions, and further suggest that his advice applies to AI middleware as well.

While a review like this one can provide some insight into the capabilities of a particular AI middleware product, nothing compares to the developer taking the time to evaluate the product of interest. All of these vendors offer an evaluation version of their products that the interested game developer should explore. Likewise, candidate vendors should be examined for the level of sup-

port that they offer. All the vendors presented in this review were available for questions by e-mail and phone, however their level of documentation did vary in quantity and quality.

To make sure the AI middleware is the right tool for the job, developers must fully define and understand the AI needs of their game. Such a self-evaluation makes the difference in choosing which vendor may provide the best solution. For example, if the developer needs pathfinding support, then the developer would require additional in-house support to use an AI middleware product that lacked this feature.

As more games begin to market higher-quality AI to customers, be it in the form of a player assistant or companion, monsters to fight, or a programmed opponent, AI will become more of an important consideration to the game development community. AI middleware offers some potential solutions. 🐼

Visit Gamasutra.com to read expanded reviews of each of these products, as well as more AI middleware products available to game developers.

VENDOR SHEET

- **AI Implant**
Biographic Technologies
Montreal, Quebec, Canada
(514) 844-5255 x250
www.ai-implant.com/games
- **DirectIA**
MASA Group
Paris, France
U.S.: (212) 343-8838
www.directia.com
- **Renderware AI**
Criterion Software
Austin, Tex.
(512) 478-5605
www.renderware.com/renderwareai.html
- **Simbionic**
Stottler Henke
San Mateo, Calif.
(650) 655-7242
www.simbionic.com

Life under the Satellites

Mark Webley on Lionhead's approach to multi-project management

Not only has Peter Molyneux's Lionhead Studios created innovative game ideas like *BLACK & WHITE*, the studio has also forged new ground in development and resource management with its satellite studio idea. Lionhead wanted to help young, independent development teams by looking for ways to keep a development company's creativity intact while offering them Lionhead's resources. The satellite scheme's mission was to allow outside groups to keep ownership of their companies, with Lionhead taking a modest stake in them — from both a financial and managerial point of view.

In 1999, Big Blue Box became the first Lionhead satellite developer. The second, Intrepid, joined soon after. 2001 saw the creation of Black & White Studios, and a new outfit is developing *THE MOVIES*, a standout of originality at this year's E3.

Game Developer had a chance to talk to Lionhead's development director Mark Webley about Lionhead's unique approach to game development and its possible impact on the gaming community.

Game Developer: What is Lionhead's biggest challenge right now?

Mark Webley: I think the major problem is trying to get a game out within a reasonable timescale while maintaining really high quality thresholds.

If we could sit down at the beginning of a product cycle and say, "This new game is going to be done in two-and-a-half years' time," and then two-and-a-half years later, boom! the game's finished — and the game is good — we'd be laughing.

We are learning all the time, and the arena for making games is changing all the time. Core team sizes are generally much bigger, there's a lot more expected of a game [from the consumer side], and the production values are much higher than five years ago. Today, there's a much wider diversity of skills required to put a game together.

Game Developer: One of Lionhead's goals since its inception has been to keep a small, lean team of 20 to 25 employees. Have you kept that goal?

MW: I would say that these figures are reasonably true even today, as we have grown Lionhead in a totally different way than just adding people. Using our satellite model we've set up a number of different studios which are made up of teams of 20 to 25



Lionhead co-founder Mark Webley looks to encourage an environment that fosters creativity.

people each, who focus very intensely on developing a particular game. But currently Lionhead — as an independent organization — has around 200 employees.

GD: How much autonomy does each of Lionhead's satellite studios have?

MW: Lionhead currently has four satellite studios, and there is also the development studio based at Lionhead's main office.

In terms of day-to-day operations, the studios are very autonomous, but in terms of the development plan and strategy, these are discussed regularly. There are a number of monthly meetings whereby the studio heads get together and discuss their projects and future plans.

GD: Is this a business plan you see more studios adopting? What lessons has Lionhead learned from running this style of business model?


MW: Whether this type of model works for other studios really depends on the studio and the strength of the satellite.

All Lionhead satellites have a very strong management and production team, people that are very experienced in making games, and are staffed by people we know and have worked with in the past. Whether it works for someone else depends on their philosophy. For us, one of the reasons for keeping studio sizes small and manageable was one of culture. Culture is very important at Lionhead and something which gets difficult to manage when a company gets too big. That's something we loved when we started Lionhead and something which we have fought to maintain.

GD: Lionhead is currently working on a new concept and title, *THE MOVIES*. Can you give us a brief idea of how this concept originated and the steps taken to develop this idea into a full-fledged game?

MW: *THE MOVIES* is quite interesting in the way that the project came together. In December 2001, Peter came into the studio and said that he'd had a dream about a game about running a movie studio.

By March 2002 we had the first two team members in place, Adrian Moore and James Brown; we had worked with Adrian before and both had worked with each other in the past. We knew what the game was about and it could be summarized on a single page.

From the original idea to getting the first inkling of a team together took three to four months — unique for Lionhead. The team has just recently moved into their own offices. 

Using an Arithmetic Coder: Part 1

When creating a networked game, we need to transmit messages between a client and server, or between peers. Bandwidth is expensive on the server side and perhaps scarce on the client side; therefore we want our messages to be as small as possible while containing as much information as possible. In other words, we want those messages to be compressed.

At present, most games don't do a good job of compression. Often, games will compose network messages using values that are multiples of a byte in size; more ambitious games will trim their values down to a 1-bit resolution using something like the Bit Packer I discussed last year ("Packing Integers," The Inner Product, May 2002). But as I pointed out in that article, a 1-bit resolution still wastes significant bandwidth when your values are small. I introduced the Multiplication Packer as a simple way to pack values without wasting space.

Back then I only discussed packing, which is only one part of compression. Statistical modeling is the other part: if data contains some values that are more common than others, we can exploit that fact to reduce the overall size. That's where the Arithmetic Coder comes in.

This month, I'll provide some engineering reasons for using an arithmetic coder. I'll also talk about packing values in the absence of any statistical modeling. We'll do modeling next month.

Engineering Reasons to Use an Arithmetic Coder

The arithmetic coder is a plug-in replacement for the Bit Packer or other buffering scheme that your game already needs. I've found my engine gets simpler when I switch to an arithmetic coder, due to the coder's elegance. I also gain confidence in the engine's solidity, since it becomes impossible to have range-checking problems. Briefly, I'll explain the reason range-checking is necessary with a bit- or byte-packer.

An attack on your system, or perhaps just a fouled-up packet transmission, can cause your game to receive a message filled with garbage values. Therefore you can't trust any value you read from a network message. Specifically, suppose you are unpacking a 4-byte value that indicates the length of some array. You might know that a legitimate client will never pack a value higher than 5,000, which you chose as the maximum array length. But if you read the 4-byte quantity without range-

y=2	10	11	12	13	14
y=1	5	6	7	8	9
y=0	0	1	2	3	4
	x=0	x=1	x=2	x=3	x=4

FIGURE 1. When packing two numbers, x and y , together into a message, the number x can take on values from 0–4 and y can take on values from 0–2, producing a total of 15 possible cases, represented by the numbered grid squares. The number in each square is $y * 5 + x$, which is what the Multiplication Packer would compute if you first packed y , then x .

checking it and it consists of garbage data, you could get a number in the billions. Subsequently attempting to allocate a billion-element array will cause you problems.

For this discussion, the basic API for packing a value into a message looks like this:

```
void Arithmetic_Coder::pack(int value, int maximum_value);
int Arithmetic_Coder::unpack(int maximum_value);
```

Suppose that somewhere in my code I have defined:

```
const int ARRAY_LENGTH_MAX = 5000;
extern Arithmetic_Coder *coder;
extern Array array;
```

If you want to put the length of an array into the current message, do this:

```
coder->pack(array.Length, ARRAY_LENGTH_MAX);
```

The coder is probably written to assert if you pass a value parameter that exceeds the `maximum_value`. When the guy receiving the message wants to get the length back out, he does this:

```
int length = coder->unpack(ARRAY_LENGTH_MAX);
```



JONATHAN BLOW | Jonathan is hanging out at a coffee house. He's hungry, even though he just had a Bulgogi Burger. Send dinner suggestions to jon@number-none.com.

LISTING 1. THE MULTIPLICATION PACKER

```
Multiplication_Packer::Multiplication_Packer() {
    accumulator = 0;
    range = 1;
}

void Multiplication_Packer::pack(u32 value, u32 limit) {
    range *= limit;
    accumulator = (limit * accumulator) + value;
}
```

At this point, `length` will be between 0 and `ARRAY_LENGTH_MAX`. So long as you've defined `ARRAY_LENGTH_MAX` appropriately, you can't obtain horrible results. Garbage input will just give you a garbage value within the legitimate `range`. (Though you may wish to detect garbage packets somehow, it is now unlikely that improper detection will cause your game to crash.)

Once the arithmetic coder is in place, you can start feeding it probability tables that describe the input data, and your bandwidth usage will magically go down. Generating these tables does take a little bit of effort, but they are not required; if you are prototyping, or you don't care about certain message types, you

don't supply tables for them. It should be standard practice for networked games to use an arithmetic coder. Despite all the benefits, though, I'm not aware of a single game that uses a full-blown arithmetic coder for networking: a tragedy. (I'm using one in my current project, but it's not done yet, so it doesn't count.)

How an Arithmetic Coder Works

Lots of references out there can show you various mechanisms for implementing an arithmetic coder; see Howard and Vitter's paper in *For More Information*. Here I'll look at the coder from a nontraditional viewpoint and try to supply some less easily found intuition about why it works.

Recall the basic functionality of the Multiplication Packer (Listing 1): You call `pack` repeatedly to create a message; when you're done, you have an integer between 0 and `range - 1`, which you somehow put into a network packet. Back then we used the `Bit_Packer` to do this; the magnitude of `range` tells us how many bits we need. Visualize the packing operation as indexing a 2D grid (see Figure 1). See last year's *Integer Packing* article for more detail.

The Multiplication Packer was inconvenient; every time you pack a value, `range` gets larger. If you try to pack too much, the

packer overflows the 32-bit integer and you're in trouble. We would like to spool the data into a buffer as we pack, to prevent overflow and eliminate the cumbersome use of a separate Bit Packer. Unfortunately it's unclear how to do this; because all the bits of `range` change every time you multiply, there's nothing coherent to store in such a buffer.

The arithmetic coder gets around this using a simple but effective trick: instead of packing an integer between 0 and `range - 1`, it packs a fixed-point binary number between 0 and 1. The Multiplication Packer added information to a number by letting its magnitude grow divergently. Now, instead, we add information by growing the number to the right of the decimal, in such a way that it converges toward a limit. Because the number converges quickly, its most significant bits become stable after each packing step; those stable bits can be written into a buffer, freeing up space within the 32-bit integer.

It's astonishing how easy it is to make this change when playing with the math. The Multiplication Packer gives us a number in the interval $[0, \text{range})$ and we want a number in $[0, 1)$ so we just divide by `range`.

If we write out the function of the Multiplication Packer as a recursive rule, we get this:

$$\begin{aligned} \text{accum}_k &= \text{limit}_k \cdot \text{accum}_{k-1} + \text{value}_k \\ \text{range}_k &= \text{range}_{k-1} \cdot \text{limit}_k \end{aligned}$$

After n total packing steps, `range` will be:

$$\text{range}_n = \prod_{k=1}^n \text{limit}_k$$

Now I am going to rewrite the equation for accum_n by expanding the definition of accum_{n-1} :

$$\text{accum}_n = \text{limit}_n \cdot (\text{limit}_{n-1} \cdot \text{accum}_{n-2} + \text{value}_{n-1}) + \text{value}_n$$

We can repeat this expansion recursively, rewriting the term for accum_{n-2} and so on. I'll do it just once more:

$$\text{accum}_n = \text{limit}_n \cdot (\text{limit}_{n-1} \cdot (\text{limit}_{n-2} \cdot \text{accum}_{n-3} + \text{value}_{n-2}) + \text{value}_{n-1}) + \text{value}_n$$

We refactor this equation by distributing the multiplication across the addition:

$$\text{accum}_n = \text{limit}_n \cdot \text{limit}_{n-1} \cdot \text{limit}_{n-2} \cdot \text{accum}_{n-3} + \text{limit}_n \cdot \text{limit}_{n-1} \cdot \text{value}_{n-2} + \text{limit}_n \cdot \text{value}_{n-1} + \text{value}_n$$

Now an interesting pattern starts to become clear. But the preceding equation still isn't fully expanded, because we have that $accum_{n-3}$ term that we could keep expanding all the way down to $accum_1$. Let's do that:

$$accum_n = \frac{\prod_{k=1}^n limit_k}{\prod_{k=1}^1 limit_k} \cdot value_1 + \frac{\prod_{k=1}^n limit_k}{\prod_{k=1}^2 limit_k} \cdot value_2 + \dots + \frac{\prod_{k=1}^n limit_k}{\prod_{k=1}^n limit_k} \cdot value_n$$

Recall that I am doing all this because I want to divide by $range_n = \prod_{k=1}^n limit_k$. And looking at the equation now, it's trivial. This equation wants to be divided by $\prod_{k=1}^n limit_k$. It's begging for it. So:

$$\frac{accum_n}{range_n} = \frac{value_1}{\prod_{k=1}^1 limit_k} + \frac{value_2}{\prod_{k=1}^2 limit_k} + \dots + \frac{value_n}{\prod_{k=1}^n limit_k}$$

The terms of this sequence converge, because the denominator grows geometrically but in general the numerator doesn't grow. The result is the value between 0 and 1 computed by an arithmetic coder. To reiterate, it's the value computed by the multiplication packer, divided by $range_n$. This divide allows us to save out the most significant bits of the result incrementally, achieving successful packing of arbitrarily many values.

To me, the grid-ness of the multiplication packer is easy to visualize, but the shrinky-ness of the arithmetic coder is less so. Being able to factor between them gives me comfort.

Arithmetic Coders and Security

As we've seen, when we use an arithmetic coder to pack up messages, individual data items get multiplied by arbitrary values before they are written into the output buffer. To a casual viewer — someone looking at your network transmissions with a packet sniffer or a hex editor — the data will appear unstructured, since important fields will not land on bit boundaries. Since your game protocol is difficult to see, it's difficult to hack.

This statement is more than anecdotal when you look at the situation from an information-theoretic viewpoint. Compression works by exploiting and reducing predictable structure. This increases the "entropy" of the data. Data with no structure whatsoever is random and has maximum entropy. Thus, perfectly compressed data appears completely random.

This idea of maximum entropy comes up in another area we're familiar with: encryption. Like compression, encryption is about crunching on data in a reversible way to produce maximum-entropy output. So in a sense, perfect compression


is equivalent to perfect encryption; the probability tables act as a secret key.

Unfortunately, our current compression schemes are nowhere near perfect, so data that's "encrypted" via compression is very crackable. Even high-order statistical models of the input data leave a lot of structure in the output, so you should not use just an arithmetic coder to encode life-or-death secrets and then consider them secure. My point is that when you use an arithmetic coder, hacking your protocol becomes a matter of employing statistical analysis, known plain-text attacks, and the like — either that, or reverse-engineering all your networking from assembly language. Both of these options require substantial effort on the part of a would-be hacker; most of the people with enough knowledge to hack your protocol will be off programming their own games.

So just by using an arithmetic coder, with our primary goal being to save bandwidth, we also raise the barrier to entry for those who want to hack our game. That's a nice side benefit.

Now suppose you do really want to secure your data stream. You should use a hard encryption algorithm for this. But even in that case, the arithmetic coder helps you out. The reason is that hard cryptographic algorithms become easier to brute-force attack the more you know about the input data. Suppose a hacker is playing your game and types a chat message; this causes an encrypted network packet to be sent to the server. The hacker knows that the source data is mostly ASCII text, and he or she can use this knowledge to help break the encryption key. But if you compress the data with an arithmetic coder prior to encryption, the hacker must work a lot harder to break the key. For more of an explanation of this, see section 8.5 of the sci.crypt FAQ listed in For More Information.

Charles Bloom pointed out to me that the preceding discussion is slightly dangerous, since there have been several attempts to use arithmetic coders as strong encryption, but all such systems have been breakable. So I want to re-emphasize that I am not encouraging the use of such schemes. If you need strong encryption, use a strong encryption algorithm. If you don't need strong encryption, you can still take comfort in the fact that by compressing your data, you've gained protection against the casual intruder.

You can download this month's source code at www.gdmag.com. 

FOR MORE INFORMATION

Howard, Paul G., and Jeffrey Scott Vitter. "Practical Implementations of Arithmetic Coding."

<http://citeseer.nj.nec.com/howard92practical.html>

MACM-96 Multi-precision Arithmetic Coder Module

www.cbloom.com/news/macm.html

Cryptography FAQ for sci.crypt. section 8.5: "How do I use compression with encryption?"

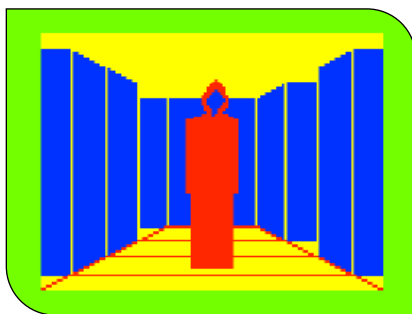
www.faqs.org/faqs/cryptography-faq/part08

Genre Art, Part 2: Third-Person Games

If you work in the game industry, chances are that you yourself enjoy playing games. I know that there are some who work in the industry because they couldn't get a gig with Pixar, and some are just killing time until they move to their first-choice career (I actually worked with a programmer who wanted to be a landscape gardener), but most professional game developers have played games fairly consistently from around the time they left the womb. As this is the case, most of us have lists of games that represent the very highest points of our gaming experience over the years.

Admittedly, there is some nostalgia involved when we gaze back through the Vaseline-smeared lens of our mind's eye to the games we feel most impressed us, but allowing for the technological limitations of the time, it is often fair to say the best games always manage to capture our imagination in some way.

Last month ("Genre Art, Part 1," July 2003), I looked at first-person games, and I am reminded of what was probably my earliest experience of a genuine first-person game (even though at the time, I had no idea that that was what I was playing). The game was called PHANTOM SLAYER, written by Ken Kalish, and during the summer of 1982 I sat terrified in my friend Mark's living room (he was the one who always had the cool computers), staring at the achingly garish colors of the Dragon 32 on the screen and waiting to be caught by the titular Phantom. In this game, which was in hindsight far ahead of its time, the player negotiated a simulated three-dimensional maze, controlling what was essentially the camera, thus providing a first-person experience.



The early simulated-3D game PHANTOM SLAYER began to shed light on the role of player perspectives in games.

Remembering this game and how distinct it was from the majority of other games of the time made me aware of how much more prominent the third-person game has been through the decades. No doubt the technical limitations of early machines made the third-person perspective an obvious option, but placing the player on-screen also strikes me as a conscious design decision which in many cases must have surely related to the building and marketing of a character-based brand.

Here is the center of the third-person genre's defining quality: A third-person perspective structures a game largely around character. There are some exceptions, but most third-person games have players control a character that is obviously not themselves (despite the rather lame process that has occasionally sur-

facied of allowing players to import their own likenesses into the game). The experience for the player of playing the game is a projection of what the character experiences on the screen.

But before I get bogged down in trying to determine what the psychology of the third-person experience tells us about the human condition, let me focus for a moment and bring myself back to the purpose of this article. Seeing as many of us will now or at some future time be involved in producing art for a third-person game, what do we need to think about or avoid in order to be successful in this genre?

Environment

So what do we need to be aware of when creating a third-person environment? As there is such a huge variety of game styles that fit under the banner of "third-person," it's impossible to address all of the possibilities. A generalization may be helpful: Think about what the player will see.

This is a rule that game artists must follow regardless of the specifics of their project. Failure to give this rule proper consideration inevitably wastes time and resources in areas that will ultimately have little or no impact on the player's experience.

The issue of clarity, or readability, is important for third-person games. One



HAYDEN DUVAL | Hayden started work in 1987, creating air-brushed artwork for the game industry. Over the next eight years, Hayden continued as a freelance artist and lectured in psychology at Perth College in Scotland. Hayden now lives with his wife, Leah, and their four children, in Garland, Tex., where he works as an artist at 3D Realms. Contact Hayden at haydend@3drealms.com.

major (if obvious) distinction between first- and third-person games is that players in a first-person game have no on-screen presence (not counting weapons or hands), and as such are extremely limited in terms of how they move through the levels. Some jumping and crouching is fine, but more adventurous types of movement are hard to convey in the first person and so they are also likely to be confusing and not at all fun.

Third-person perspective, however, is ideal for complex character movement and interaction with the environment, as the player has a good view of exactly where he or she is and is also able to appreciate the quality of their character's animation. Hence players are best served by an environment that gives them a clear visual understanding of where they can go and what they can do.

Hazards that take the player by surprise because they look identical to the rest of their surroundings are never a good idea, as players feel cheated when they're not given an opportunity to avoid them. Visual cues need to balance between the obvious and the invisible so that a player has to be vigilant but not clairvoyant. Hazards such as lava are fairly easy to read, but as artists we have failed a player who runs across a surface expecting it to be safe because it looks like most of the others, when it is in fact deadly stinging Devil Grass.

Landmarking is also as important in a third-person game as it is in a first-person one. While assets are likely to be duplicated around a level, anything that helps players navigate areas of exploration will help reduce the chances of player frustration that arise when they feel lost.

If the third-person camera presents us with a reasonably distant point of view, much of the detail in the environment can be shifted from the geometry into texture where appropriate. However, we are at a point now where chances are that we will run out of texture memory well before our polygon count becomes an issue, so geometry detail can be a better solution. This is especially the case with lighting systems that produce dynamic shadow-

ing, where texture detail remains static as opposed to the shadows generated by features that are actually modeled.

Larger exterior areas can also benefit from some form of procedural texturing to provide tonal variation, with a detail layer to make it look like grass or dirt. There is also the option of including vertex coloration to affect the hue in a controlled way.

Characters

Central to any discussion about the art for a third-person game is the process of character design. A third-person game by definition means that the player-character is visible (and usually centered on the screen), so character design deserves prominent consideration.

Lara Croft made it obvious that hitting the jackpot with a lead character could open doors to unimagined wealth, and if you were very lucky indeed you might get to meet Angelina Jolie. Once the link between character and potential returns was clearly established in the minds of those who signed the checks, producing the next hit character started to become more about marketing budgets than pure design considerations, but nevertheless, characters still don't make themselves, they need to be created.

Given that third-person characters spend a great deal of their time with their back to the player, it is important to put sufficient effort into the design of the character from the back as well as the front. Nowhere are a character's buttocks more important than here, but it is also wise to add secondary movement with some sort of physics-based clothing or hair geometry to break the monotony of a rigid mesh.

Since third-person game players don't project themselves into the game as they are inclined to do in a first-person game, successful characters must appeal to the player in some way to draw them in.

The Hot Chick. Playing to the lowest common denominator of the target audience is one way that has proved successful in the past. Lara may be the most celebrated example of designing a lead char-

acter whose physical attributes are more *Baywatch* than Indiana Jones. Since player demographics inform us that there are more males playing games than females, giving your audience some of what they want is good business.

Comic books have long led the way in terms of unfeasibly proportioned heroines, and creating a female character that feeds off these same stereotypes is not hard. But unless she has some kind of distinct visual identity, she is likely to be nothing more than the equivalent of an



TOMB RAIDER'S Lara Croft exemplifies the Hot Chick character.

E3 booth babe, which is to say, nice to look at, but ultimately forgettable.

How to make your Hot Chick character stand out is a matter of taste, opinion, and fitting her into your overall game design, but it is no longer the case that simply replacing the muscle-bound male lead in your action game with a sexy female vixen will gain you any points for originality. Ms. Croft and her legacy have put an end to that.

The Crazy Creature. Especially common to the platform end of third-person gaming, the Crazy Creature character is often used to solicit the player's affections. Whether it's Crash, Sonic, or Spyro, this kind of character design typically targets the younger audience in particular. Watching a few hours of kids' TV on a Saturday morning will introduce you to many of the same types of characters.

Larger-than-life personalities that match the frenetic gameplay in many platformers are underlined with the exaggerated features and over-the-top animations of this kind of character. Using an animal is a good start (even if it's something as unlikely as a blue hedgehog), as many cartoon characters have proven how appealing this route can be. If you have exhausted all your conventional animal options, there is always the fantasy animal route, best illustrated by the Pokemon pantheon of creatures that are close relatives of animals that we know, each with specific character traits and special powers.

A major advantage of Crazy Creatures is that they don't need to be even vaguely realistic in terms of movement and animation — in fact, the exaggeration of their characteristics adds to their interest.

The Tough Guy. Possibly the most boring lead character type, but in many cases,



Duke Nukem is the archetypal Tough Guy character.

he is still the best choice. To some extent, the male player is invited to project himself onto this kind of character in much the same way that we all wanted to be the Karate Kid in 1984 (although the thought now of ever wanting to be Ralph Macchio fills me with terror). In this case, the cooler the character, the bigger the payoff.

Choosing this kind of lead happens most often when working with a more realistic, mature-oriented game, so the visual design of such a character is restricted to some extent by the need for realism. The same rules that were in effect for the Hot Chick character hold true for the Tough Guy: without a striking design, he will be readily absorbed into the ranks of leading male characters from across the years. This can actually enhance the players' ability to project themselves into the game, if that's your design goal. The faceless Space Marine could quite easily be you or me, but there is only one Duke Nukem.

Life in the Third Person

Working on a third-person game is more likely to allow an artist the chance to create larger spaces and more complete worlds than a first-person game, which typically concentrates on more confined enclosed spaces. It is also a great opportunity to focus on a character design that will in many ways drive the game. Whatever the situation, it's the artists' task to make the most of what they've got. 🎮

The Space Between: Efficient Use of Downtime

Time management — it can define your success as a sound designer, recordist, or composer. Develop good time-management skills and reap the benefits of smoother projects, achieved deadlines, and less stress during crunch time. We've all heard this lecture before. But what about "the space between"? How should we be spending our time between projects or even during those slow days? Here are a few productive alternatives.

Computer maintenance. If you are like me, you have a love/hate relationship with your machines. You love them when they work and hate them when they don't. Downtime is the best time for dealing with disk fragmentation, file backups, driver updates and software installs. Remember, a happy computer is a working computer.

Technical maintenance. Take some time to inspect cables and connectors. Dust off those soldering skills and make any necessary repairs. Properly install any gear that might have been hastily implemented during your last project. If you are using a patch bay, take the time to clean the patch cables and jack field. Make sure that the patch bay labeling and studio wiring documentation are updated.

Sound effect and sample libraries. Update your databases. Regardless of whichever system you use to keep your sound effects and samples organized, take advantage of downtime to keep them current. Assets recently created for one project could prove to be invaluable for another in the future. Taking the time to catalog and document those elements now can save a lot of headaches and hair pulling later. Nothing is worse than digging through piles of backed-up audio sessions looking for the full-bandwidth version of that killer explosion you created nine months ago.

If you are not currently using an online solution for sound-effect storage and

management, you might want to take the time to investigate those options. Software solutions such as SoundLog Pro, NetMix, MTools, and Soundminer all offer databasing, search, conversion, and export tools for multiple workstations sharing a common library. CDEExtract is a similar software package for auditioning, cataloging, and converting synth sample libraries.

Resource gathering. If you have a Rolodex, make sure you use it. If you don't have one, then go get one. Use this time to fill it with contact information for current and potential vendors, contractors, musicians, and voice actors. The worst time to be searching for any of those people is amidst a project with a looming deadline.

Training. This can be as involved as taking a class or seminar on a very targeted subject, or as simple as brushing up on a few of the manuals you have been using as a doorstop for the last six months. I have accidentally stumbled upon a variety of amazing features on gear that I use every day by perusing the manual. Also, download demos and evaluate new audio software applications. In addition to keeping current with new technology, it helps you to reevaluate the tools you are currently using.

Consider joining a trade-related organization. Groups such as the AES (Audio Engineering Society), GANG (Game Audio Network Guild), and the IA-SIG (Interactive Audio Special Interest Group) provide online resources, discounts on products and seminars, and a wealth of contacts through other members.

Listening. Take some time to critically evaluate the sound on current games, tel-

evision, cinema, and albums. This goes beyond just brushing up on the competition and keeping up with the Joneses. As an artist, exposure to other professionals' works can be an inspiration as well as an education. It is important to look beyond the game industry. Sound design and music provides influence across all forms of media. Styles and trends found in film audio and composition can easily be applied to interactive entertainment.

Exploring. Experiment not only with your creativity but also with the technology with which you surround yourself. Under time constraint we often resolve to using proven methods when developing content. Take the opportunity to reinvent the wheel. Sometimes the new wheel is faster, more efficient, and sounds better.

Productivity in the studio is a by-product of our own creativity and the tools we use to capture it. The space between gives us the time and opportunity to maintain those tools and refuel our imagination. 🛠️

RESOURCES

ONLINE SFX & SAMPLE DATABASES

CDXtract: www.cdextract.com

MTools: www.mtools.info

NetMix: www.net-mix.com

Soundlog Pro: www.soundlog.com

SoundMiner, Inc.: www.soundminer.com

ORGANIZATIONS

Audio Engineering Society (AES)

www.aes.org

Game Audio Network Guild (GANG)

www.audiogang.org

Interactive Audio Special Interest Group

(IA-SIG) www.iasig.org



MIKE VERRETTE | Mike is audio director for Wicked Noise and a member of the Game Audio Network Guild. During periods of downtime he reads everything you send him at mike@wickednoise.com.

A Tale of Two MMOGs

This month I'm doing a spot check of two upcoming massively multiplayer online titles that through a mix of social interaction and internal mini-games, are both trying to reach audiences beyond the hardcore gamer. It's interesting to see how well each follows three of the earliest-published rules from The 400 Project.

THERE. The first of the two titles is **THERE** (www.there.com). There Inc. started back when dot-coms were still booming and secured over \$30 million in funding, which they've put to good use, hiring some heavyweights of game development, like Stewart Bonn, one of EA's first producers, and Amy Jo Kim, one of the top experts on online communities. They've also taken years to get their online world going and are rumored to be taking many more months of tinkering before they're ready for a full launch. **THERE** provides an online world with very realistic avatars depicting young men and women, in a sort of cyberspace Club Med setting, complete with fashion shows, trivia contests, scavenger hunts, hover boards, paint guns, and the option to design and sell your own clothing and other items.

TOONTOWN ONLINE. **TOONTOWN ONLINE** (www.toontown.com), brought to us by Disney, is a massively multiplayer world for young kids, where they can chat using preselected phrases and participate in both 3D arcade mini-games as well as a larger overall simple quest-oriented structure not unlike that of many standard MMORPGs. There is combat of sorts against Cogs, grim robotic NPCs and their buildings, but it is combat via joke and gag, and players accumulate new types of gags as they level up.

Rule: Maintain Suspension of Disbelief. Both games do fairly well with this rule. **TOONTOWN** sets up a world that, though different from our own, feels internally consistent and very Disney — it's immediately comfortable and cheerful. **THERE** has a more ambitious task, creating a



TOONTOWN ONLINE is geared toward providing online entertainment for young kids.

world closer to reality and aiming at a wider range of players, but it also manages to do well at creating an inviting, more grown-up world. One quibble: you start the game with \$10,000 in "Therebucks," but you arrive almost naked, and a T-shirt costs about \$1,500, with other clothes priced accordingly. Since the world seemed so realistic, that immediately roused my disbelief — I would much rather have been given \$100 at the beginning with T-shirts costing \$15, or call the currency "beads" that could reasonably be 1,500 to the shirt. **TOONTOWN**'s currency supports this rule better, though based on jellybeans. Who knows how many jellybeans it should take to buy a squirting-flower gag?

Rule: Provide Clear Short-Term Goals. Here, **TOONTOWN** leads the player elegantly but unobtrusively from the start. **THERE** players are left to flounder at times, but of course they are farther from launch and are likely to provide more directed activities as their user base grows. I expect they'll depend more on a friendly user base helping newbies get up to speed,

which is already starting to happen.

Rule: Provide Parallel Challenges with Mutual Assistance. Both games are fairly evenly matched on this rule. Both provide multiple ways to earn their respective currencies and gain skills, and gaining currency or skill in one activity can make others easier, providing mutual assistance.

Both online worlds incorporate good design rules and have clearly profited from analysis of previous online titles. Whatever luck they have opening the online world to both kids and a wider range of adult players will benefit all online game developers in the long run.


From the mailbox. This month, a couple of responses to my June 2003 column on consistency, "The Hobgoblin of Little Minds."

Brett Douville of LucasArts says, "It's my belief that you have stumbled across a meta-rule in discussing the Emerson quote. This is, in fact, a rule about rules. Another way [of] stating it might be: 'Rules should be followed, except when they shouldn't be.' It is often the role of experience to determine when and why rules should be broken; there is no greater teacher of wisdom than experience."

While Major Jeff Bourne of Texas A&M University suggests that the important thing is internal consistency, even if the rules change within a game. He refers to my example about exploding crates: "Players don't need to know that boxes explode; however, once they start to learn that they explode, there should be some reason why they do, and some way to alter destiny based upon their actions."

More comments on consistency will follow in future columns. ✍

NOAH FALSTEIN | Noah is a 23-year veteran of the game industry. His web site, www.theinspiracy.com, has a description of *The 400 Project*, the basis for these columns. Also at that site is a list of the game design rules collected so far, and tips on how to use them. You can e-mail Noah at noah@theinspiracy.com.



From the Ground Up: Creating a Core Technology Group

Illustration by Claudia Newell

CLINTON KEITH | Clinton is the lead programmer for the Engine and Tools Group at Sammy Studios, located in Carlsbad, Calif. He's worked in the game industry for nine years, formerly as the lead on *MIDTOWN MADNESS 1*, and as project director for Angel Studios.

It's not every day a studio gets to reinvent the game development wheel by setting up a brand-new technology group. Combining awareness of tomorrow's trends with the knowledge gained from game development's past, Sammy Studios chose to invest in a new technical infrastructure.

Hardware and technology are changing rapidly. Game development teams and budgets have grown to keep pace. However, our methods for creating game technology have often not kept pace. We waste millions of dollars and years of time to create games that don't meet the gameplay or visual quality bar that consumers expect.

In many cases, this situation is due to methodology that we use to create game technology. Ad hoc methods used to support a few artists or tune a simple game mechanic are not suitable for teams of 30 people or more. Also, given the growing size of programming teams, ad hoc leadership and team communication styles are no longer effective.

The most apparent effect of these problems is an increasing amount of thrashing, or wasted effort within a team. Content creators (artists and designers) are often delayed by key technology; even if they do have a working pipeline, they are delayed by long iteration time between making changes to assets and seeing them in the game. On the programming side we wind up chasing frequently changing goals and conducting death-march efforts to keep pace with schedules created from some dim, optimistic past.

These are not the best conditions for creating the games we wish to create. Such conditions leave little time for exploration of what will make the game fun. At worst, they sap the passion of the developers to make a great game.

The game development industry needs to mature. It needs to develop technologies and methods for freeing up creative and content roadblocks. It needs to keep schedules realistic and provide time for refactoring our code and updating our assumptions.

To that end, this article describes building a technology foundation for a new game development studio. It's about taking all the team members' accumulated experience and trying to get it right from the start. It's about addressing specific technical infrastructure problems that can prevent us from making the best games possible. Although it is far easier to start with a new group and blank slate, every one of these problems described can be addressed by any existing programming team. The solutions presented are based on our collective experience over our careers and may be ideal for your organization, but the goal is to present a starting point. Many of our lessons learned have come from failure as well as success.

The Goals

Our goal is simply to make our development teams as productive as possible. We want to give the content creators (designers and artists) fast and intuitive control through tools which allow them to discover gameplay and resolve production problems as early as possible. We want to establish a methodology that will allow the programmers to work in an effective team environment. Our focus is to create the technology and processes that will support these goals and provide the basis for a number of development projects running in parallel. The approach we've taken is to invest heavily in a technology infrastructure from the start. This meant creating a sizable Engine and Tools Group from the beginning.

The following article describes the decisions we've made regarding technology, tools, and methodology. It's about our solutions to common problems given the opportunity to start from nothing but a commitment by Sammy Studios to invest in a technical infrastructure.

Technology

Technology is the foundation for development. We want to architect this foundation to make it both flexible for prototyping and robust for production.

Data-driven design. Game design requirements are very dynamic, and our technology needs to be designed to handle this. Game behaviors and tuning parameters must be iterated frequently to produce the best results. Game engines often do not support this approach. A common practice is to embed the behavior of the game's entities too deeply into code. As a result, a programmer will end up spending a great deal of time making small code changes and building a new version of the game for a designer. To address this, programmers might create simple text-format files for storing frequently changed parameters, but don't make the parser robust enough to handle format changes and backward compatibility.

Another problem is depending too much on object hierarchies for behavior. Anyone who has written a large object hierarchy knows that moving object behavior around the hierarchy can produce a great deal of problems in the long run. An exam-

ple of this is moving AI behaviors around the hierarchy until you end up with AI behaviors in base classes or a great deal of cut-and-pasted code. Both of these solutions create a fragile code base that becomes increasingly difficult to maintain.

A data-driven design can solve these problems. The system that we created is called an Actor Component System. This system allows groups of components, or basic objects of behavior, to be aggregated together to form actors within our games. The components that make up actors are driven by XML data files which the designers or artists tune with a Maya plug-in editor. Components and actors communicate with each other through a messaging system that allows the data contained in the components to be loosely coupled.

For example, say you have a locked door. The designer may want you to change that door to have it unlock when there is a specific class of NPC in view, which would require adding an “eye” component to the door. When the eye component “sees” an NPC it recognizes, it broadcasts a message to its parent actor indicating that the door should unlock and open. The benefit of this approach is that you don’t have to have all door objects contain eyes, and eyes don’t have to know to what they are attached. A simple scripting system glues the logic together (for example, seeing a particular NPC would trigger a door-open action). Making this change in an object hierarchy behavior model would be more challenging.

There are problems to be aware of with a data-driven design. You can easily give the designers too much control or provide too many controls to adjust in this system. This can result in unforeseen combinations of behaviors that can generate a great deal of problems. We address this issue by having programmers combine components into Actors templates ahead of time. You don’t want to create a system that attempts to remove the programmer from the design loop.

Middleware. As a new studio, middleware was an obvious choice for us. Halfway through the current console cycle is not the best time to be creating a new technology base. Creating your own core technology requires a time-consuming process of hiring specialists in programming graphics, physics, audio, and all the rest, for all platforms. The amount of time it takes to create this core engine adds a great deal of risk to development.

We chose to leverage mature middleware wherever possible, which has accelerated our prototype development. Middleware vendors provide plug-ins and exporters for Maya or Max, allowing us to focus programmers familiar with the SDKs for these programs on extending functionality for our own use.

Middleware must be carefully evaluated. We’ve rejected some middleware packages after our evaluation determined we could not meet our goals with them. Middleware that does not have

TIP: Focus your interface development in areas that will be used the most. A value that is rarely set can be given a text field. A frequently tuned value may deserve a custom slider control.

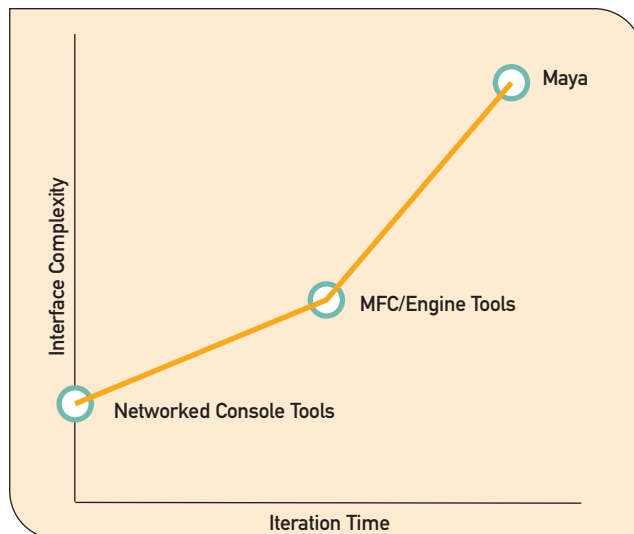


FIGURE 1. Three-level tool hierarchy based on iteration time, depth of data manipulation, and interface complexity.

source code licenses adds a great deal of risk and has been a big basis for not using certain libraries. Middleware that has not been used on a published game is also a risk. Such risk might be acceptable if you were replacing some existing technology, but in our case we didn’t have technology to fall back on. Also, some middleware can be suited for prototyping but not for production.

Engine design. People are often confused by our effort to develop an engine after we have chosen to use middleware. Such confusion stems from misunderstanding what a game engine really is. It’s not a renderer, but rather a framework and wrapper for the various subsystems (including middleware) in the application. It unifies resource management, control, sound, networking, and gameplay with common interfaces that allow them to work well together.

Engine design is often neglected, which can lead to problems. When middleware and platform implementations are not well insulated, replacing them can create major headaches. Subsystems that are not insulated from one another can create a web of cross-dependencies that build up during development and take more and more of the programmer’s time to maintain. When subsystem interfaces are created independently, it becomes anyone’s guess as to how systems will all work together properly.

The solution is to architect the engine and framework as early as possible. For us this began with an agreement about the coding standards and project structure. A design phase defined the top-level design, the interfaces, and several use cases describing the top-level flow of a game using it. Our framework consists of a number of subsystems that inherit an identical interface. This interface defines each phase or operation of the subsystem from startup, reset, simulation update, rendering, and shutdown (among others). These subsystems are treated as tasks with their own priorities. This framework

TIP: Having a single interface where everyone on the team can control the version of their assets and launch the game from one place is very useful. In the past I have written independent tools to do this, but we were able to integrate this tool into the Alienbrain client. Using the Alienbrain instant messaging system can bring everyone on the team immediately up-to-date with any changes to the assets, executable, or exporters. Links to new files can be sent and automatically updated.

allows us to control the game flow at this highest level of code rather than having lower-level systems having to “know” about each other.

Insulating the higher-level code from the lower levels is important. This includes creating wrappers or defines for middleware-specific types and isolating platform specifics through common interfaces. Proper interfaces are the key to solid engine design.

Generic networking libraries. Online networking is a popular feature these days, and it’s important to address it early, as it’s not easy. Leaving network development to later in the project will create a lot of refactoring in your game object behavior. These objects need to be developed and tested with networking technology in place.

We created a generic network layer very early and have benefited in many ways. It allowed us to test new behaviors in the networking environment as soon as they were written and fix problems that are best solved when the code is fresh in the mind of the author. There were also a few surprising benefits as well: By allowing early network play, our designers had early insights on potential AI behavior. In addition, we have fully leveraged this technology for our tools, creating robust tools that run on the PC and work and communicate with the games running on the consoles.

Tools and Pipeline

Our goal is to give content creators fast and simplified access to the technology. The more times they can iterate and the less time spent waiting for fixes, exports, and programming changes, the better the game will be.

Tools. Tools for development are essential, but their development can easily be mishandled. They can limit content quality and production flow if not properly developed. This is often due to limited resources being dedicated to tool production early on. Tool development can also be too ambitious, providing complex, deep tools that do not meet the expectations of professional artists who are used to mature interfaces. Tools can place a major burden on users by introducing complex steps or latency between creating assets and seeing them in the game. They might depend on parallel functionality in the game

that could be changing rapidly and end up requiring heavy maintenance. At worst, an asset that works in the tool might not work at all in the game once it is imported.

Our approach to tools is to create them at three levels. These levels correspond to how tightly coupled the assets and data the tool manipulates are with the game (and how fast changes happen) and how deep the user interface is. Figure 1 shows the relationship among the different types.

The top level consists of plug-ins and extensions to Maya and other commercial tools. Maya has hundreds of man-years of development in its user interface, there is a rich pool of talent that knows how to use it, and its interface is extremely customizable and extensible. This is what the artists and designers use to perform the large-scale operations of creating levels and geometry and setting up gameplay. They spend most of their time in this environment, and so their tools need to be solid.

The mid-level tools are MFC applications linked to our engine. An example of this is a tool that allows us to create and tune our animation finite state machine (FSM). FSMs are often defined in code or in obscure text files that designers or artists cannot manipulate. A user interface such as MFC allows your tool programmers to create capable interfaces rapidly. The artist is graphically manipulating the FSM and its parameters, and can see the immediate progress within the game view. There is nothing that can be lost in translation or code duplication between the engine and the tool when they are linked together.

The lowest level of tools we develop are those which run directly on the console. These tools manipulate data that is dependent on the hardware. An example of this is our mip-map tuner which allows the artist to select a texture within the game running on a PS2 and tune the mip-map (*l* and *k*) settings in real time. The networking layer allows this tool to be run on a PC that is on the same network as the PS2. Once the artist is happy with the settings, he or she saves the parameters out to the asset pipeline, which uses those values for all subsequent PS2 exports.

Another important feature is to make sure the interfaces are as uniform as possible, for example, making all your camera controls work the same as Maya’s. Keeping a dozen or so tools under proper control creates problems. Properly versioning the tools and the assets they create is a major requirement for the pipeline, which I’ll address next.

Asset pipeline. Channeling the flow of thousands of assets (source and exported) through a system that maintains many revisions can be a major challenge. The problems are too numerous to list individually, so I’ll generalize them:

- Maintaining revision control not only of the assets but of the executables. When everyone has different versions of the

Renderware Graphics 3.5 introduced platform-independent XML format (RF3) and export templates that control how the RF3 is converted to platform-dependent assets. This is a great example of an intermediate file format that made the pipeline far more extensible.

game, it's hard to track down problems but easy to lose tuning improvements.

- Old assets that are no longer useful clutter the system long after they should be retired.
- Numerous paths for adding assets exist. No permission system exists to protect the data.
- No meta-data exists to control the asset export. For instance, what would you do if you needed to change the scale of every exported asset?
- Bad data (assets that can crash the game, for example) needs to be caught before it goes out to the rest of the team.

The first step in creating an asset pipeline is to visualize what you want it to do. We flowchart the path for assets through each system we want to create and work with the artists and designers to develop case studies of how specific areas of a pipeline will work. The goal of this flowchart is to identify and remove bottlenecks for the artists to create scenes and see them in their final form.

Many developers have created custom asset management tools that required major investments. The impact on budgets and schedules due to bad asset pipelines certainly justified the expense. However, there are some recently released commercial applications that make such an investment in homebrewed solutions no longer necessary. We chose Alienbrain as our base asset management system. Alienbrain came with Maya integration built-in and an extensive COM-based interface that allowed us to integrate it with our engine and tools.

One other key element of the system is the use of an intermediate XML file format that is exported by the tools (Figure 2). This intermediate file format is an additional file that is exported into the pipeline. It contains all the data that you would potentially be interested in. This gave us two major benefits:

First, assets can be re-exported from an automated system if we wish to change some basic value. For example, when we wanted to rescale our geometry, we changed one float in one template and hit one button to re-export everything.

The other benefit is that exported assets can be deleted and regenerated every night. Together with meta-data-driven asset tracking, this is a useful system for culling old assets that are no longer used.

THE BEST PRACTICES DOCUMENT

During development there are many improvements to the technology or methods made to make life easier. How do you track these improvements? A single document that collects descriptions of these is a great help.

Think of what you would want to hand to new programmers joining your team. You want them to come up to speed on your team's practices as efficiently as possible. If they need to know it, it should be in the best practices document.

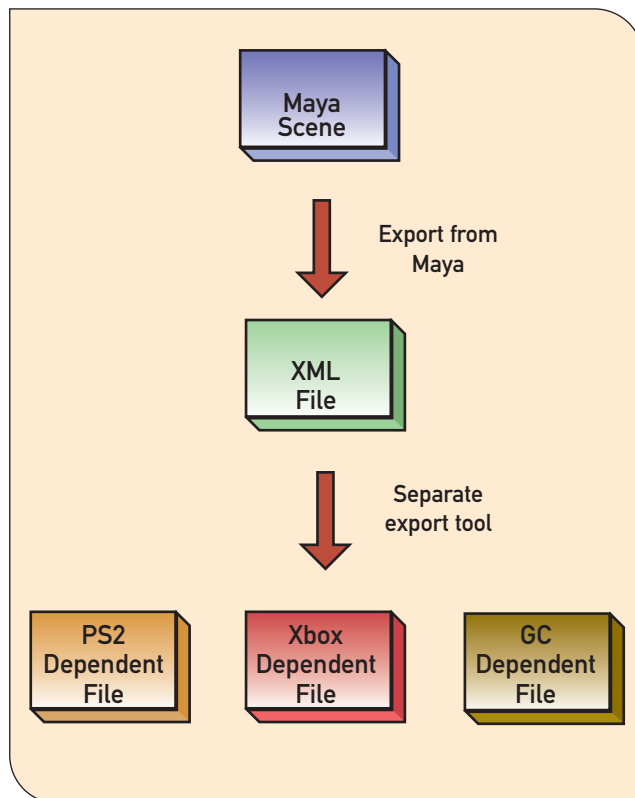


FIGURE 2. Three-level tool hierarchy based on iteration time, depth of data manipulation, and interface complexity.

The major ongoing issue of an asset pipeline is that it is constantly changing. With the addition of tools during development it is easy to introduce problems and pathways that make it harder to use. Revisiting the state of the pipeline and fixing problems is a must, as asset pipelines are never truly “finished.”

Methodology

The effectiveness of a programming team is determined by how well they are organized and how well they work together. A team that is not moving toward a commonly understood goal and sharing the same practices is not going to be very productive.

Shared practices. Creating a game is a team effort that should be supported by certain practices. Such practices include sharing tool improvements and improving the ability for programmers to understand each other's code. Code that is hard to read, poorly documented, or full of bugs hinders efforts to streamline programmer productivity. Improvements to the technology and development tools need to be shared widely enough to benefit all programmers on the team.

To help solve this, we use a best practices document. This document is a collection of all the standards and practices that

have been established. This document is constantly updated to include improvements or refinements to the system.

The best practices document includes coding standards, setup instructions, naming conventions, documentation requirements, commit practices, and descriptions of useful utilities and tools for the programmers. Revisions to this document happen continually; whenever someone sends out a useful macro for debugging, I'll have them include the information for that in this document.

A best practices document alone is not sufficient, however. Other practices such as code reviews and pair programming exist to ensure that the entire team is continually following these practices and that good practices are being promoted. If these practices are followed, you'll find that everyone's code quality will improve and maintenance will be reduced. Programmers write better code when they know more people are going to look closely at it.



A lot of automation to control the commit practices can be built. This automation makes the task of committing and sending mail to the team as painless as possible.

Commit practices. Source version control tools are essential, but they can introduce as many problems to a project as they solve. It's very easy for programmers in a rush to commit untested code changes that bring the entire project to a halt. It's not uncommon to see teams spending half their time fixing problems related to this issue.

We've set aside a PC that is our commit test target. Before a programmer makes a commit, he or she first reserves this machine. Following the commit, the test PC retrieves those changes and rebuilds all configurations of the game. When all the builds are successful, the target PC is released and the programmer sends a note to a team list describing the changes. This catches most of the problems committed, but not all of them. Daily build tests catch many of the rest.

Nightly builds. A common problem occurs when you're not sure what version of the game or assets is being used by members of your team. An artist might have a crash problem on his or her machine, but the problem cannot be replicated on a development system. Trying to figure out such puzzles wastes a large amount of time.

Earlier I mentioned that we re-export all of our assets overnight. This is done on the PC that is used as the commit test target. The tool that creates these builds also embeds version numbers in the executables and the game (for run-time version testing). Each morning the assistant producer runs the game that was regenerated overnight and goes through a regression test. Any problems must be solved immediately. Once a working set of assets and executables are identified, they are copied up to a network drive. Everyone on the team is

informed (using Alienbrain instant messaging) that they can update to these versions.

The benefit of this is that the team can copy known working assets and executables to their local drives and start making changes. If an artist introduces a new asset that breaks their local copy of the game, then they know they caused it and that that they cannot commit this new asset. The same goes for programmers changing code. In such a situation the artists are encouraged to seek a programmer to solve the problem.

Leadership. Programming teams are often led by someone who does not yet understand how to lead. That person has shown a great talent for programming and was probably promoted with no instruction on how to fill the lead programmer role. This situation can lead to disaster for the team, because the lead will continue to focus on programming and not leading the team.

Leads need to spend half their time managing the effort, dealing with problems that are affecting the team, planning to avoid future problems, and making sure everyone is working toward the same goal. During milestone crunch times, they will need almost all of their time free for putting out fires. As a result, leads should not assign themselves key tasks around critical milestone deliverables. Leads should focus on mentoring and taking a global view of the technology being developed by the entire team. There is no way a lead programmer alone can create enough useful technology that would offset the benefit gained by having someone focusing on team issues.

From Investment to Returns

Many of the problems described here are common to every developer. Our solutions were developed based on our current circumstances and collected experience. These same solutions may not apply to you, but the problems still need to be addressed. Creating and justifying the expense for infrastructure can be an uphill battle with management; the value added by infrastructure cannot easily be tracked by counting games sold. A solid infrastructure does not ensure a hit game; rather it reduces the number of obstacles that get in the way of creating a hit game. 🐛

SUGGESTED READING

- Brown, William J., and others. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.
- Cerny, Mark, and Michael John. "Game Development Myth vs. Method." *Game Developer* vol. 9, no. 2 (June 2002): pp. 32-36.
- Game Programming Gems*, vols. 1-3. Charles River Media, 2000-2002.
- McConnell, Steve. *Rapid Development*. Microsoft Press, 1996. (Or any of McConnell's other books.)
- Meyers, Scott. *Effective C++, 2nd Ed.* Addison-Wesley, 1997.

Data-Driven Subsystems for MMP Designers: A Systematic Approach

Corporate IT departments have long leaned on relational databases as the foundation for their mission-critical applications. There are many reasons why they chose this route, but one of the most significant has to be because relational databases are robust, reliable, and in short, they just plain work.

This has not been lost on massively multiplayer (MMP) game developers; many are using databases to implement the customer billing, account management, and character persistence aspects of their online services. However, designers of MMP games (or any game with very large content requirements) can also leverage the power of a relational database to implement data-driven game subsystems effectively.

The advantages to the game designer are many. For one thing, relational databases can build a more maintainable, extensible, and flexible game, qualities that can ultimately make a game better. The game can be tweaked or grown in a large variety of ways simply by modifying existing data values, or adding new data to realize new content.

In addition, data-driven designs can utilize the easy-to-learn yet powerful Structured Query Language (SQL) to write queries that answer any question imaginable about existing game content in the database. This reduces the need to find answers in potentially out-of-date design documents or by searching through source code.

Referential integrity in the database can also serve as an extra set of eyes watching out for errors in data entry that violate assumptions made in the game. These rules prevent the occurrence of bugs that might otherwise be expensive to track down and fix if they got into the game.

Furthermore, choosing a commercial database makes the operations department very happy, because the game is built on top of enterprise-quality software, making their job significantly easier compared to home-grown solutions. Using tools that come with the database, they can easily perform on-demand and scheduled backups and restorations as well as monitor database activity and disk space usage.

Finally, databases provide easy access to in-game information

for community staff, allowing them to leverage the data for additional offerings that increase player interest and loyalty, while they also allow a more dependable offering at launch, increasing the possibility of having a life outside of work.

Getting Started

There are two major parts to the puzzle of data-driven design. One part is determining how to write game code to utilize the data within a database. This is beyond the scope of this article, but I've included some resources that address this topic in the For More Information section.

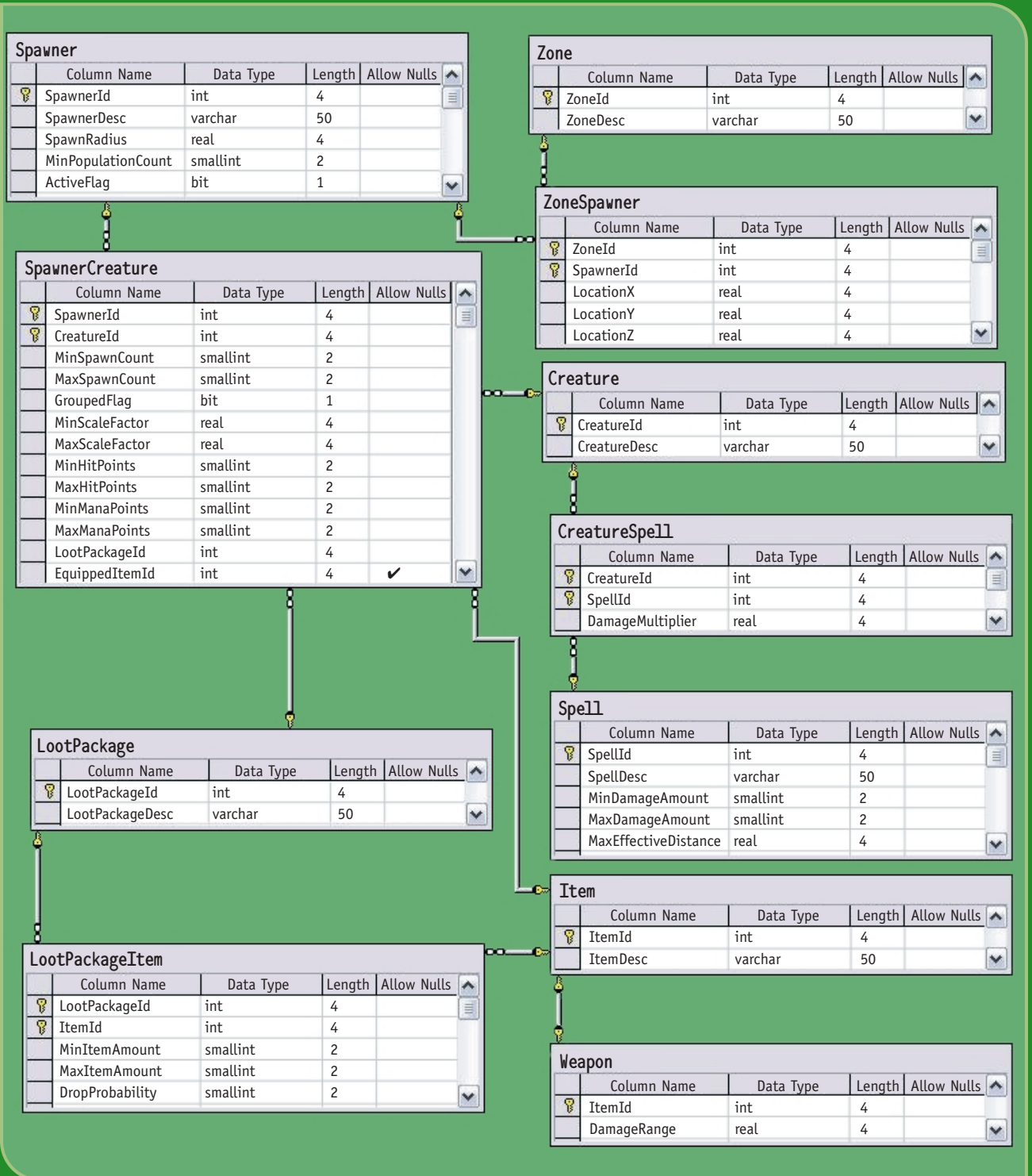
The other part of the puzzle is the creation of the database structure, or schema, that will drive the game. The remainder of this article describes an easy-to-follow process to create the schema needed to support a game's data-driven subsystem.

First, I'll name and define the steps in the process. Next, I'll introduce a real-world and nontrivial example subsystem and describe its requirements. The result will be a database schema that fully supports these requirements. Then I'll discuss the different roles involved, providing an overview of the various responsibilities of team members during the process and once the schema is implemented. Finally, I'll offer some direction on how to go about applying this approach to your game.

Before jumping in, be aware that some assumptions have been made when presenting the material. You should already have some basic understanding of relational databases and how they are structured into tables with columns and rows that contain data. You should also understand basic SQL syntax and how it is used to retrieve and manipulate the data in tables. If this is not the case, check out the For More Information section for suggested sources for this background information.

JAY LEE | *Jay recently completed his eighth year in the game industry after spending 10 years wearing suits to work. His body starts to twitch at the thought of going back to a dress code. Jay is currently working on TABULA RASA for NCsoft Corporation and can be reached at jlee@ncaustin.com.*

FIGURE 1. Database schema for creature-spawning subsystem



The Process

There are four steps in the process of creating data-driven subsystems: (1) discover entities, (2) discover relationships, (3) assign attributes, and (4) iterate.

The “discover entities” step uncovers the things that we care about in the subsystem. These are the objects, actions, or concepts considered to be in scope of what the subsystem is to accomplish. For example, a **Spell** is an entity in a spell-casting subsystem, while an **Item** would be an entity in a vending subsystem.

In step two, “discover relationships,” we consider the entities from step one and ask whether they share an association of importance that is within scope. If they do, we determine the nature of the relationship. In a spell-casting subsystem, the **SpellBook** entity has a relationship with the **Spell** entity. A **SpellBook** entity can contain many **Spells**, and the same **Spell** can appear in different **SpellBooks**.

Additional pertinent information about entities and relationships is captured in step three, “assign attributes.” **SpellName** is an example of an attribute of the **Spell** entity in a spell-casting subsystem.

The last step, “iterate,” requires revisiting steps one through three as many times as needed until stasis is achieved. The process is deemed complete when additional iterations do not reveal any new entities, relationships, or attributes, and the participants agree that what is captured meets the subsystem requirements.

Now that we are armed with an overview of the process, let’s look at our example subsystem.

Creature Spawning Requirements

An effective way to learn a process is to follow along as it is being applied to a familiar yet nontrivial example. Most MMP games contain creatures of some type, and these creatures have to be made to appear in the game world. The subsystem that does this is usually known as the creature-spawning subsystem.

The following requirements were identified for a creature-spawning subsystem:

First, spawner objects may be placed anywhere in the 3D world and are utilized to create creatures for players to battle. The valid spawn area is anywhere designated by the radius of a circle from the given location of the spawner.

The second requirement is that the spawner should ensure that a minimum population of creatures is maintained for its area and should respawn as needed when this falls below the desired level as creatures are killed. The distribution of creature types by a spawner should be configurable.

Third, the game world is divided into zones, and each zone represents a geographic region that is allocated to a server. When the server running a zone starts up, it should place and initialize all spawners assigned to that zone. Spawners can be tagged as inactive so that they are ignored at the next server restart.

Fourth, spawners can spawn multiple creatures from the set of creatures; each spawned creature will be assigned random hit points, mana points, and size based on allowable ranges. Multiple instances of a type of creature may be spawned at a time, within a range. If they are tagged as grouped, the creature manager should be notified so that the appropriate group AI kicks in.

Fifth, when creatures die, they distribute loot to the player that landed the killing blow. The loot carried by a creature should be assigned from a specification that allows for a range representing the random amount of the given item, and a probability out of 1,000 that the item is actually dropped (like a die roll). Creatures may drop multiple different items. Players get experience points equal to the hit points of the creature divided by the number of people in the party.

The final requirement is that creatures are armed with a single (optional) weapon. When in range, they utilize the weapon in combat when they are unable to cast any spells they have. However, when they die they do not drop the weapon. Creatures may have multiple spells to use in combat but have the same set of spells regardless of where they are spawned. They only configurable attribute of a creature-based spell is a factor representing the damage done compared to the base spell damage. For example, a factor of 1.5 means that when used by the creature, the spell does 1.5 times the normal damage.

It may not be the case that all our requirements fall out as neatly as these have been defined, but that doesn’t invalidate the process, nor does it prevent us from starting to develop our schema. The steps are always the same; we might have to employ more iterations before arriving at a satisfactory result if the requirements come about in a more piecemeal fashion.

Let’s begin developing the schema with the first step in the process.

Step 1: Discover Entities

From a database perspective, entities are the things that we care about representing in the subsystem. They are typically the nouns used by designers when defining requirements, either verbally or in a design document. Entities should be labeled with singular names that are clear in intent. From the requirements we can quickly identify **Creature**, **Spawner**, **Item**, **Weapon**, and **Spell** as candidate entities. Some less obvious entities might be **Zone** and **LootPackage**. The latter is a term we’ll use to describe the specification of items dropped by creatures.

Every entity should be given a primary key whose numeric value is automatically assigned by the database. A primary key is an identifier capable of uniquely addressing a particular instance of an entity. In Figure 1, the **Creature** table shows **CreatureId** as the primary key on the **Creature** table. We do not care specifically what its value is, as long as it is unique and remains unchanged once assigned. For example the Orc may have a **CreatureId** of 25, whereas the Wolf may have a **CreatureId** of 12.

Each entity should also be assigned a description or name. This is the familiar handle assigned an entity instance by which it is normally referred. However, it's not the primary key because variable-length strings are relatively inefficient to query, and we may at some point want to rename the creature currently referred to as "Orc."

Our list of candidate entities is **Creature**, **Spawner**, **Item**, **Weapon**, **Spell**, **Zone**, and **LootPackage**. Each would be given a primary key and a description. This could be recorded in a word processor, on a whiteboard, or using a tool suitable for capturing database schema.

Step 2: Discover Relationships

When we consider how entities relate to each other, this allows us to start developing a picture of what the subsystem is really about. When taken alone, **Spawner** and **Creature** can offer little. Once they are related, however, we observe that they can describe what types of creatures are spawned by a particular **Spawner** or that a particular **Creature** is exclusive to a single **Spawner**.

Relationships or associations are implemented in terms of placement of the primary key of entities onto the tables of the other entities in the subsystem. In some cases the association of two entities results in a third table, forming a new entity with a compound key, a key with more than one column.

Figure 1 shows that the association of **Creature** and **Spawner** has resulted in one such new table, not so subtly named **SpawnerCreature**. This table has a compound key made up from **CreatureId** and **SpawnerId** and should be interpreted to mean that in order to address a single row on this table, one must provide both a **SpawnerId** and **CreatureId**.

To uncover relationships in a subsystem, we must ask for every pair of candidate entities: For any given entity A, how many are there of entity B? The same question is posed in reverse: For any given entity B, how many of entity A are there? For **Creature** and **Spawner** we are asking: For any given **Spawner**, how many **Creatures** may be spawned? Then, for any given **Creatures**, how many different **Spawners** may spawn them? The answers to these questions reveal the type of relationship used to associate the given entities.

If the questions don't make sense for a pair of entities, or if the answer to both questions is 0, we can be fairly certain that no relationship exists within the scope of the subsystem.

If the answer to both questions is 1, then they are in a 1-to-1 relationship (1-1). This type of relation is relatively rare. It captures a significant constraint. In the example, it would mean that a **Spawner** can only spawn one type of **Creature**, and that a particular **Creature** may only be spawned by a single **Spawner** instance in the entire game. It's unlikely to represent what is desired in the subsystem. But if it were, the **SpawnerId** column could be placed on the **Creature** table, or vice versa, to represent the association. The foreign key must only allow unique values to ensure the 1-to-1 nature of the relationship.

Tips for Leveraging the Schema in the Game

WHICH LIBRARY? Retrieving the data for use in the game is obviously critical. Be diligent when selecting the library used to interface with the database. Choices run the gamut from vendor-proprietary to industry standard, from easy-to-use to arcane. We must understand what we are choosing and why and ensure that it is extremely reliable in delivering the data into the game.

AVOID THE SERVER GAME LOOP. It is extremely beneficial to avoid directly accessing the database at critical points in the server game code, since database calls block while processing. Preload and cache the data for run-time use. Load player data only when they log in and implement an asynchronous access strategy for anytime we need to hit the database.

NOT DIRECTLY FROM THE CLIENT. The client is too speed-sensitive to directly access the database. We can generate data files for use by the client offline from the database and retain the benefits of centralized repository of data.

OPTIMIZE THE STRUCTURE FOR RUN-TIME ACCESS. When the data is needed at run-time, it must be accessed as efficiently as possible. Map the loaded data into constants, lists, hash maps, arrays, or whatever choice returns what you need most effectively for any given set of data. These choices are independent and unaffected by the database structure, which is exactly the flexibility we want.

USE PYTHON? If Python is used on your game, you must check out a product called mxODBC. This multi-platform Python library makes database access incredibly simple and supports a large array of database products.

USE REFERENTIAL INTEGRITY. Utilize RI in the database to help keep bugs out of the game. Use it everywhere possible. If a bug shows up because bad data is entered, write new rules to detect the error and prevent it from happening again.

USE STORED PROCEDURES. Stored procedures wrap SQL statements and control statements into network-efficient, reusable database objects that can take arguments. They abstract schema specifics and can keep code immune to schema changes. Not using them should be a criminal offense.

USE MICROSOFT SQL SERVER. Take advantage of the Database Diagram, a very useful object in SQL Server that allows you to build and maintain your schema visually.

If the answer is 1 to one question, and "many" to the other, the entities are associated in a 1-to-many (1-m) relationship. The "many" answer reveals the entity that receives the primary key from the other entity as a column to signify the relationship. The inherited column is known as a foreign key from the sending entity. If a **Spawner** can spawn multiple **Creatures** but a **Creature** could only be spawned by a single **Spawner** instance, **Creature** would get **SpawnerId** placed on it. If a **Creature** could be spawned by many **Spawners** but a **Spawner** could only spawn a single **Creature**, **Spawner** would get the foreign key.

FIGURE 2. A SQL query that tests for *Spawners* that violate a 30-meter *SpawnerRadius* in a specific *Zone*.

```
SELECT SpawnerDesc, SpawnerRadius, LocationX, LocationY, LocationZ
FROM Spawner A, ZoneSpawner B
WHERE A.SpawnerId = B.SpawnerId AND ZoneId = 5 AND SpawnerRadius > 30
```

If both answers are “many,” then a third type of relationship known as many-to-many (m-m) has been found; a *Spawner* can spawn many *Creatures* and a *Creature* can be spawned by more than one *Spawner*. This is what is called for in the requirements, resulting in the creation of the *SpawnerCreature* entity and its table. The foreign keys from both entities in a many-to-many relationship are brought together to create a new table and act as its primary key.

Running through the candidate entity list and asking the questions, we might arrive at the following relationships and types: *Spawner-Creature* (m-m), *Zone-Spawner* (1-m), *Creature-Spell* (1-m), *LootPackage-Item* (1-m), *Creature-LootPackage* (1-1), *Item-Weapon* (?-?). It’s a first pass, and for now we are not even sure what type of relationship *Item* and *Weapon* are involved in, only that they somehow go together. This is O.K.; perhaps as we add some detail to each entity in the next step, things will become clearer.

Step 3: Assign Attributes

The third step is to assign attributes to entities. Attributes are additional columns added to a table to describe further the entity to which they are assigned. An attribute should apply completely and only to the entity that it is describing.

For *Spell*, that might include columns such as *SpellDesc*, a description of the spell; *MaxEffectiveDistance*, a value indicating the distance from the caster where the spell is no longer effective; and *MinDamageAmount* and *MaxDamageAmount*, the range of hit points of damage delivered by the spell. *CreatureId* would not be an attribute, since the same *Spell* can be associated with more than one *Creature*.

We look through each of the requirements defined previously, and when we find what looks like an attribute, we assign it to the most likely entity. Adding or removing attributes is fairly trivial, so we don’t let ourselves get too hung up in this step. Look at some of the entities in Figure 1 for additional examples of attributes gleaned from the requirements.

Step 4: Iterate

Once a single pass is made through the first three steps, it’s time to examine the resultant schema to determine whether it is complete and correct. As new information is discovered or better understood, we make the appropriate modifications and iterate again. This is done as many times as needed, until we reach a point of stasis. We may go back to the require-

ments one more time to validate that each one is addressed, or present the schema to other team members to see if they have anything to add.

Let’s iterate through our current results and see what happens. After some consideration, it is apparent that we will want to reuse *Spawners* across *Zones*. If a particular *Spawner* is effective, then it is beneficial to assign the same instance elsewhere in the game world. The *ZoneSpawner* table results from this decision. The actual location of the *Spawner* is recorded on this table also.

The association of *Spawner* and *Creature* appears solid. We’ll add some attributes to *SpawnerCreature* to validate it. *MinSpawnCount*, *MaxSpawnCount*, *GroupedFlag*, *EquippedItemId* (and other attributes seen in Figure 1) all appropriately describe a *Creature* created by a *Spawner*.

Because we discover that it is desirable that a given *Creature* have different loot depending on where it is spawned, we will add *LootPackageId* as an attribute (and foreign key) to *SpawnerCreature*. This renders the original association between *Creature* and *LootPackage* obsolete, so it is removed.

LootPackageItem captures what is needed; the attributes *MinItemAmount*, *MaxItemAmount*, and *DropProbability* allow us to specify easily what is needed for creatures to drop loot.

CreatureSpell also appears solid. The attribute *DamageMultiplier* can be added to it.

Now to our special case, the association of *Item* and *Weapon* that puzzled us. It turns out that in our game, every *Weapon* is an *Item*, but the reverse is not true. This is known as a dependent relationship, because having an entry on the *Weapon* table is dependent on having an identically valued entry on *Item*. To represent this type of relationship, we carry the key of *Item* onto a new entity called *Weapon*. *Weapon* is implemented as a separate table and would have its own weapon-specific attributes (for example, *DamageRange*) while sharing the attributes of its row on *Item*.

Figure 1 in its entirety represents a very solid pass at the database schema needed to support the requirements of the sample creature-spawning subsystem. As an additional exercise to help verify completeness, we may want to consider asking questions of the schema and see how well they can be answered.

For example, suppose the *Zone* with the *ZoneId* of 5 is known to contain terrain that is tricky to navigate. Can we determine whether there are any *Spawners* that violate the recommended radius of 30 meters for tricky terrain? Figure 2 shows an SQL query that easily provides the answer, including giving the location in the world where we can find the offending *Spawners* should we desire to do a visual check.

Try other scenarios yourself. You should discover that the schema stands up to just about any question that you can imagine to ask.

Armed with an understanding of the process itself, let’s move on and consider the various roles of the team members developing data-driven subsystems.

Roles

During schema development, the designer primarily acts as customer; that is, he or she is considered the domain expert for the subsystem being developed. The designer defines the requirements that the subsystem must meet, both via a design document and in person to clarify details.

When the designer plays this role, the programmer is an analyst and database administrator. In this role, he or she gleans the salient requirements by reviewing the design document and interviewing the game designer(s) responsible for the subsystem.

An effective approach is for the programmer to develop a straw man of the database schema that he or she believes meets the requirements as defined in documentation. The programmer then uses the time with the designer to verify that this correctly captures the requirements, recording the modifications as they are identified.

Once the schema is agreed on, it is implemented in the database by the programmer. At this point the roles of both participants change.

The designer switches to become data provider, responsible for populating the database with the values that correctly capture the game content. If the subsystem in question manages in-game items, for example, the designer provides information such as item value, quality, name, and enchantment level.

The programmer moves on to providing the data entry tools needed to enter and modify the data. This may be as simple as making available tools provided by the database vendor. For example, Query Analyzer and Enterprise Manager are data entry tools that come with Microsoft SQL Server. Or the programmer may build custom forms for data manipulation, employing tools such as Microsoft Access or Visual Basic.

While customized forms make it easier for the designer to enter data, additional development time is required. This means that there can be a significant delay between the time the schema is implemented and when it is populated with data. An effective approach is to allow data entry immediately with off-the-shelf tools, and then to build a customized front end to make the task easier where the volume of data entry justifies a custom tool.

Of course, the other critical role of the programmer is to build the game logic that uses the data entered in the database.

Wrap Up

The next logical step is to select a medium-complexity subsystem that you are working on and apply the process. Whatever you do, don't forget to iterate. When a candidate schema has been determined, implement it in the database and build some data entry tools and queries that utilize the schema. Ask yourself questions about the subsystem and experience how well what you have built answers those questions.

After that, get the subsystem implemented in the game so that

it utilizes the data that resides in the tables in your schema. Then catch yourself grinning as you change behavior in the game by modifying data, or add content to it by adding new rows to tables.

As you apply the process to each new subsystem that gets implemented, the entire game becomes data-driven. Once the team gets the hang of the process, existing subsystems created before the paradigm shift can be quickly migrated. Don't be surprised to find yourself wondering how you ever did it any other way.

I am interested in getting feedback via e-mail from those of you who choose to use the concepts presented here in your game. As an incentive, I am offering to answer any questions in the correspondence related to implementing the process in your first data-driven subsystem. That's a pretty good deal, so I look forward to hearing from you. ✍

FOR MORE INFORMATION

INTRODUCTORY DATABASE RESOURCES

Date, C. J. *An Introduction to Database Systems*, 7th Ed. Addison-Wesley, 1999.

Lee, Jay. "Relational Database Management Systems Primer," *Massively Multiplayer Game Development*, edited by Thor Alexander. Charles River Media, 2003.

Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. *Database System Concepts*, 4th Ed. McGraw-Hill, 2001.

<http://db-book.com>

COMMERCIAL DATABASE PRODUCTS

Interbase: www.borland.com/interbase

IBM DB2: www-3.ibm.com/software/data

Microsoft SQL Server: www.microsoft.com/sql/default.asp

Oracle: www.oracle.com

Sybase: www.sybase.com

OPEN SOURCE RELATIONAL DATABASES

MySQL: www.mysql.com

Postgress: www.pgsql.com

DATA-DRIVEN PROGRAMMING RESOURCES

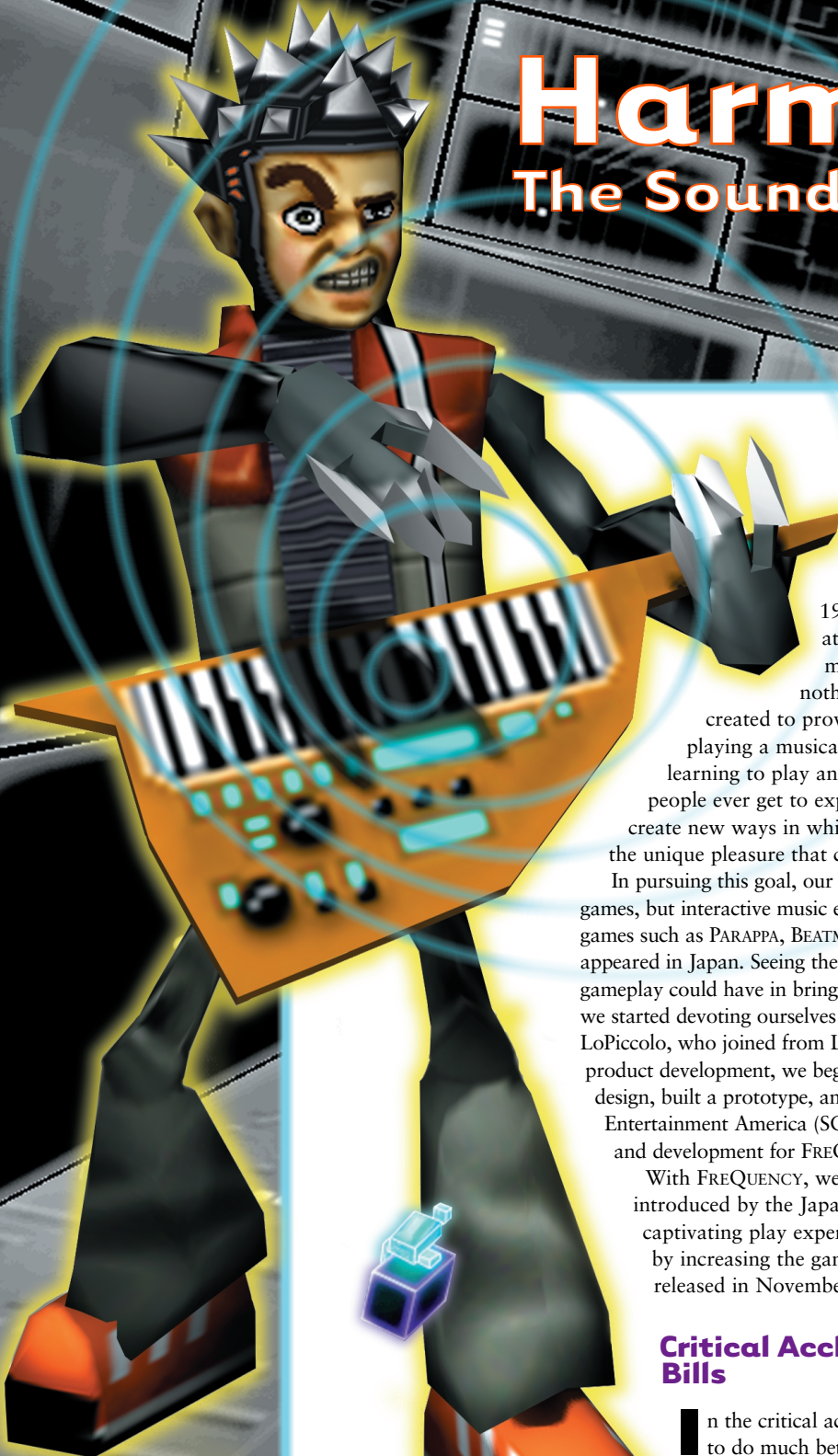
Lee, Jay. "Leveraging Relational Database Management Systems to Data Drive MMP Gameplay." In *Massively Multiplayer Game Development*, edited by Thor Alexander. Charles River Media, 2003.

Rabin, Steve. "The Magic of Data-driven Design." In *Game Programming Gems*, edited by Mark DeLoura. Charles River Media, 2000.

SOURCE FOR CREATURE-SPAWNING SCHEMA

The source for re-creating the schema in Microsoft SQL Server can be found at www.gdmag.com/code.htm. Check the readme.txt for details on usage.

Harmonix's The Sound and the Fury



When Alex Rigopulos and Eran Egozy founded Harmonix Music Systems in 1995 (they met in the computer music group at the MIT Media Lab), the duo's original motivation for starting the company had nothing to do with videogames. Harmonix was created to provide solutions for a problem we had found: playing a musical instrument feels really good, but because learning to play an instrument is too darned hard, very few people ever get to experience this. With Harmonix we wanted to create new ways in which nonmusicians could more easily discover the unique pleasure that comes from making their own music.

In pursuing this goal, our first few years were spent building not games, but interactive music experiences. But then in the late 1990s, games such as PARAPPA, BEATMANIA, and DANCE DANCE REVOLUTION appeared in Japan. Seeing the enormous potential "rhythm action" gameplay could have in bringing music-making to a mass audience, we started devoting ourselves to music gaming. Starting with Greg LoPiccolo, who joined from Looking Glass Studios to head up product development, we began recruiting talent. We drew up a design, built a prototype, and showed it to Sony Computer Entertainment America (SCEA) back in spring of 2000. They "got it," and development for FREQUENCY for Playstation 2 began.

With FREQUENCY, we wanted to build upon the foundation introduced by the Japanese music games, by introducing a more captivating play experience, both by deepening the gameplay and by increasing the gameplay's musicality. FREQUENCY was released in November 2001.

Critical Acclaim Does Not Pay the Bills

In the critical acclaim department, we couldn't have hoped to do much better. The game received enthusiastic reviews

AMPLITUDE

across the board and won several awards (including a British Academy Award), and we were flooded for months with frothing-at-the-mouth e-mails from fans who couldn't stop playing it. And how about commercial success? Let's just say that sales were ... err ... disappointing.

Despite the weak sales, however, SCEA decided (remarkably and admirably) to fund a sequel and give the concept another shot. (Credit here goes to Shuhei Yoshida, Sony's VP of product development, a genuine innovator in an industry that is ever-increasingly innovation-averse.)

Presented with this second opportunity, we sat down and spent a great deal of deep thought trying to figure out what factors contributed to FREQUENCY's failure in the marketplace, in spite of the critical acclaim. There were a number of marketing factors that one could point to, but blaming marketing is a game developer's lazy way out. Proper marketing is crucial, but it's also largely beyond the developer's control, and focusing one's critical analysis outward rather than inward is a surefire recipe for failure.

FREQUENCY's Lessons Learned

All of the available evidence suggested that FREQUENCY's gameplay was rock-solid, fun, and addictive. Then why didn't it sell? We have our theories.

For starters, the musical content in the game was almost exclusively electronica, and electronica is not mainstream music. Having an artist like Paul Oakenfold write a song for the game might have helped us among the niche market of techno-heads, but it didn't get us very far with the mass market.

But beyond the musical content, which is really as much



GAME DATA

PUBLISHER: Sony Computer Entertainment America

NUMBER OF FULL-TIME

DEVELOPERS: 20

NUMBER OF CONTRACTORS:

3 (plus 4 testers)

LENGTH OF DEVELOPMENT:

15 months

RELEASE DATE: March 2003

TARGET PLATFORM: Playstation 2

DEVELOPMENT HARDWARE:

1-2 GHz CPU PCs with 256-512MB of

RAM and GeForce 2/3/4 cards,

PS2 dev kits

DEVELOPMENT SOFTWARE USED:

Visual Studio 6, CVS (with homebrew

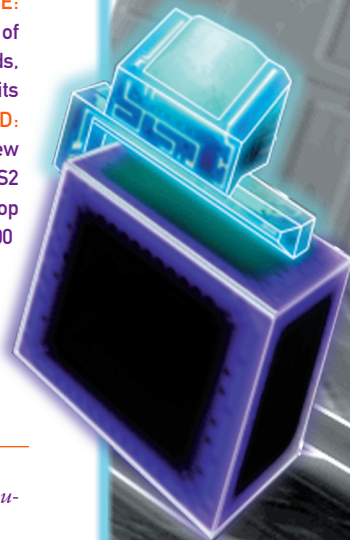
GUI), SN Systems ProDG PS2

compiler, 3DS Max 5, Photoshop

PROJECT SIZE: 380,000

lines of C++ code,

6,000 lines of script code.



GREG LOPICCOLO | Greg is the vice president of product development at Harmonix and was project director and co-lead designer on FREQUENCY and AMPLITUDE. His e-mail is greg@harmonixmusic.com.

ALEX RIGOPULOS | Alex is the co-founder and CEO of Harmonix and was executive producer and co-lead designer on FREQUENCY and AMPLITUDE. E-mail him at alex@harmonixmusic.com.

a marketing/positioning issue as anything else, we also felt, looking back, that there were some deep flaws in the game itself that impaired its success. All of these flaws carried a common theme: We didn't do enough to hook the players; not only during the opening few moments of gameplay, but before they even started playing it.

There are hundreds of games to choose from out there, and so consumers will often make their first judgment about a game based upon a momentary glance at a screenshot. If any aspect of a game fails to entice immediately, a player will move on to the next option. Critics who delved deeply into FREQUENCY and who, once inside, became hopelessly addicted to it wrote most of FREQUENCY's positive reviews. But consumers, understandably, don't have that kind of patience. We believe there were three main reasons that FREQUENCY failed to win over prospective consumers:

Not pretty enough. FREQUENCY was our first PS2 game. We scrambled to build a graphics engine from scratch within a tight timeframe and budget. Our art staff did a great job with the resources available, coming up with a cool retro look for the game that actually functioned within the constraints of the limited graph-

ical capabilities at our disposal. But at the end of the day, we didn't come close to hitting the high graphical bar evident in the PS2 marketplace. In fact our stylish, retro look often evoked the response, "It looks like a PS1 game." We knew this was a shortcoming, but at the same time we underestimated the magnitude of the impact. We had faith that, with compelling enough gameplay, looks wouldn't matter so much. We were wrong. Most prospective players will write off a game in an instant if, at first glance, it doesn't hit the graphical production standards to which they're accustomed.

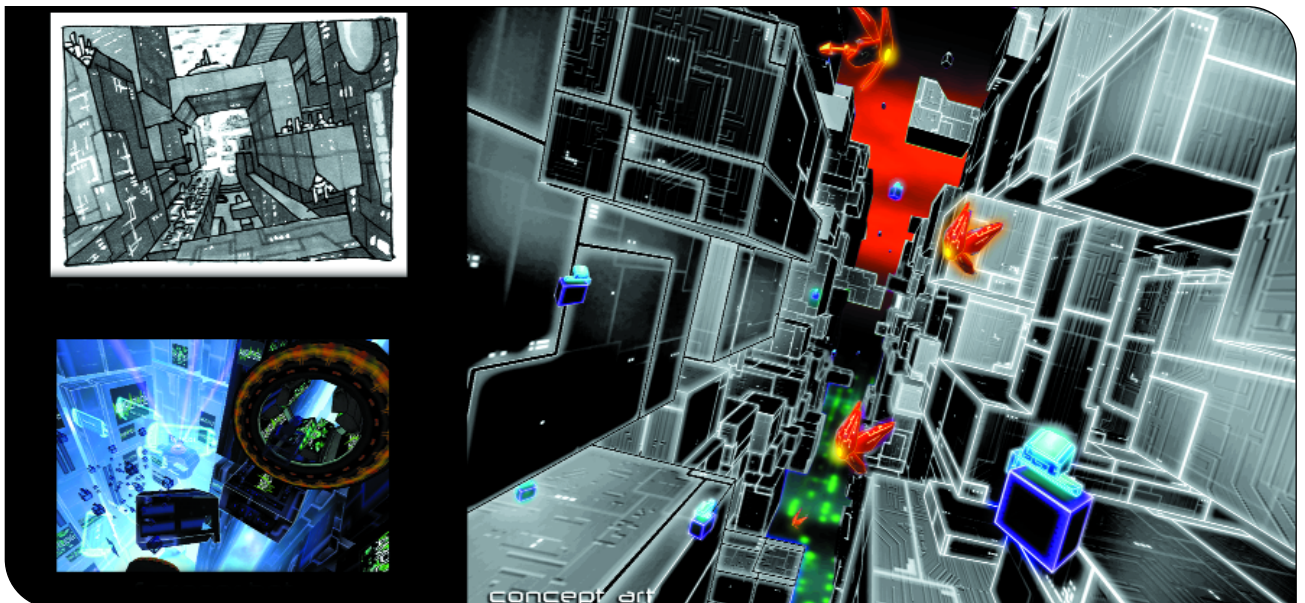
Interface too abstract. The graphical interface for FREQUENCY was not complicated — at all. It was certainly considerably simpler than the graphical interfaces of most games on the market. But our problem was that the interface was extremely abstract. With most games, it's immediately apparent from a glance at the screen what the player is supposed to do and what the game is about. If you see a football player on the screen, you know, "Hey, it's a football game." If you see a car on a racetrack, you know, "Hey, it's a racing game." If you see a brawny pair of arms holding a big gun, you know, "Hey, I'm supposed to shoot stuff." But in FREQUENCY, there is no character run-

ning around on screen, no car, no spaceship, nothing for the player to identify with. All you see are three little target spots, with blue dots flowing through them that the player must "shoot." This interface, while extremely simple once understood, is also completely foreign to most prospective players. Faced with this conceptual barrier, it was far too easy for the uninitiated to respond, "I don't get it," and move on.

Too hard. The opening game levels were simply too hard. Given its innovative interface and gameplay, what the game desperately needed was for the opening levels to be forgiving and accommodating for players who are having a tough time getting started. Instead, the struggling newbie quickly met with defeat after defeat (accompanied by humiliating boos, no less). A few more tries would have gotten them over the hump, but many players never gave us a few more tries.

Addressing These Problems with AMPLITUDE

Process. By its very nature, innovative game design carries with it a special burden: players don't know what to do at



Steps along the development path for the Metaclouds arena.



Evolution of AMPLITUDE's FreQ characters. the player's in-game musician avatar.

first. Inspired by an increased appreciation for this reality, we modified our development process for AMPLITUDE, placing an obsessive focus on a player's first impressions. Specifically, we began play-testing AMPLITUDE the moment we had a first playable, and then we repeatedly play-tested it over and over and over, every single month, through the end of development. With each successive round of playtesting, we brought in a fresh set of players who had never seen the game before. Consequently, throughout the entire development process we were getting a steady infusion of quality information about how our iterative design tweaks were affecting the way new players were reacting to our game.

Design changes. One of our first moves was to change the musical focus of the game from electronica to more mainstream music — rock, pop, metal, and rap. Chuck Doud, our producer at Sony, assembled an impressive roster of well-known bands (Garbage, Blink 182, POD, and others) and artists (David Bowie, Herbie Hancock, Pink) for the game, which helped us significantly in attract-

ing attention to AMPLITUDE among both players and music fans.

In tackling our beauty problem, we invested a much greater percentage of our production resources this time around (both on the art side and the engineering side) in making the graphics as breathtaking a part of the play experience as the music and gameplay.

To make sure new players could get the feel of the game right away, we redesigned the interface in a number of ways, introducing a spaceship that the players steer to shoot lasers at targets. This way, when a prospective player sees a screenshot of AMPLITUDE, or watches someone else play the game, he or she



Blasting a vocal track in AMPLITUDE.

immediately reacts, “Oh, I get it, I’m supposed to shoot those things.” Interestingly, this interface change really only affects players for the first few moments of play. Thereafter, the player’s attention ends up focused on the same three little target spots that were the center of the interface in FREQUENCY. But the key point here is that those first few moments are absolutely crucial for luring in the new player.

In resolving the difficulty problem, we went medieval on the tuning process. We moved from three difficulty levels to four, to support a wider range of players (especially at the bottom end of the curve), we gathered a giant pile of quantitative gameplay data at all difficulty levels from our repeated play-testing session, we built tools to visualize and analyze that data, and we kept on tuning until we knew we’d gotten it right.

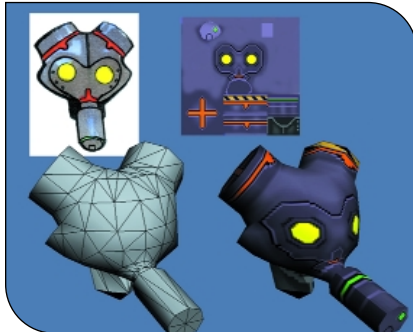
There were many other design changes — too many to mention here — all of which were similarly focused on making sure that the new player is coaxed and nurtured into the play experience from the game’s first screenshot.

What Went Right

1. Features cut early and often.

During FREQUENCY’s development, we wasted precious development time by deferring decisions to cut some features until fairly late in the development process, which resulted in the team putting hard work into features that ultimately didn’t make it into the game.

Armed with that experience (and with a hard ship date hanging over our heads), we resolved to be rigorous and realistic about the scope of our ambitions for AMPLITUDE. As each milestone was completed, we reviewed our overall feature set, and when it became clear that any given feature was not going to fit into the remaining schedule, we cut it immediately. We didn’t let hope and optimism drive our feature set, and consequently, almost everything we worked on ended up in the game, with very little work having to be discarded. (See What Went Wrong #4 for one unfortunate digression from this approach.)



Concept sketches, geometry, and skins for two final FREQ heads.



2. Lots of early re-architecture and investment in technology.

FREQUENCY was our first title on the PS2 platform and our first large-scale game. By the time we wrapped it up, we had firsthand knowledge of the many pitfalls of developing for PS2. At the onset of AMPLITUDE development, co-lead programmers Eric Malafeew and Eran Egozy made a strong case for scrapping the bulk of our core libraries and rewriting them. We wanted to take all of our “I wish we had coded it that way” moments and address them in our new project.

Our goal was to create a solid software architecture consisting of encapsulated, abstract, portable, individually testable modules. This meant that we had to wait longer than we would have liked to actually get a first playable, since the first three to four months of coding time were devoted to upgrading or rewriting our core libraries. However, this strategy paid off big-time at the end of the project, as we had comparatively few mysterious crashes, memory leaks, or other ulcer-inducing late-stage problems, and the issues that we did encounter were much more easily resolved.

3. Collaborative design process.

At Harmonix, game design is a collaborative process. As many as a half-dozen or more senior team members contributed significantly to the AMPLITUDE design on an ongoing basis. We held weekly design reviews where the design contributors met to debate design issues

(often heatedly).

With this sort of distributed design responsibility, the process had to be managed carefully and continually to keep the design from bogging down. But it was worth it; we were able to derive the benefits of widely varied design insights from team members with very different design talents and perspectives, and the result was a game design that was far superior than it would have been with only one of those individuals in charge of the game’s design. A collaborative design process also had the benefit of keeping all of the team principals bought in to the design on a fundamental level.

4. Testing, testing, testing.

Since we were building a game experience that would be new to most of our prospective audience, we were careful not to make assumptions about what players would understand or enjoy. We tested our initial design, then revised it and tested some more. We tried to interject ourselves as little as possible into the testers’ experience and thereby learned some valuable lessons about holes in the tutorials and early game flow that we would not have caught by simply explaining the gameplay to the testers. We tested each game mode (Solo, Multi, Duel, Net, Remix) separately, as many of them differ significantly in their design and user interface. When we were satisfied with the design of each mode, we transitioned to difficulty testing. By the time we shipped the game, we were pretty sure there were no unwelcome surpris-

es awaiting our audience.

5. Meeting early milestones saved us from a death march.

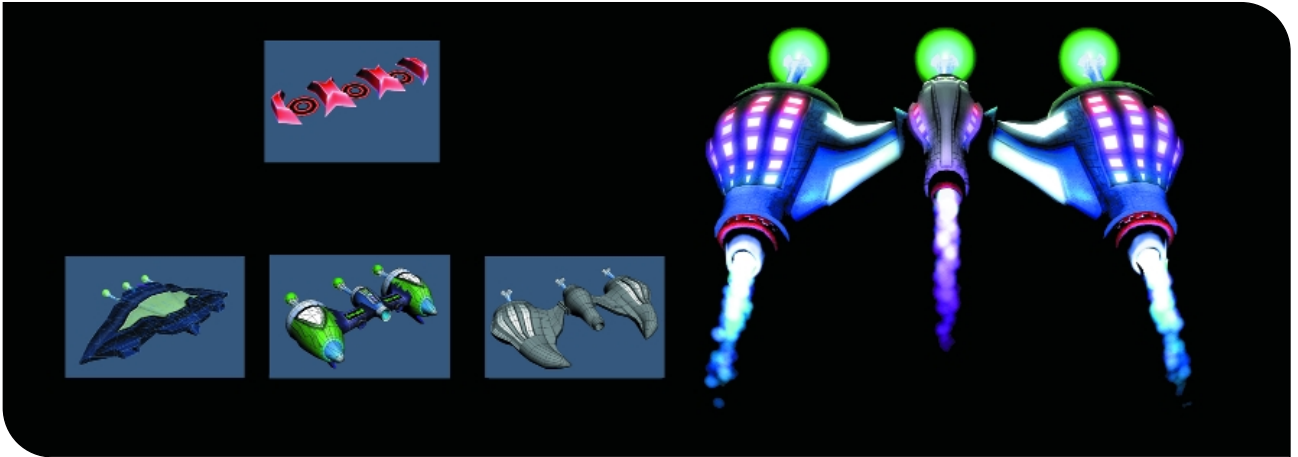
Early in the development process, we discussed as a team what sort of development cycle we wanted to undergo. By this point the team was experienced and focused, with all team members capable of assuming responsibility for their individual schedules. However, no one on the team (including management) was particularly enthusiastic about having to undergo the kind of grueling late-stage death march that we endured in getting FREQUENCY ready for market.

To minimize the risk of death march, we defined each milestone in great detail, revising and enhancing the definition as the milestone approached. As we got to each milestone date, the team stepped up its efforts to complete the milestone fully, working evenings and weekends as necessary throughout the project. Since we defined our early milestones carefully and completed them all fully and on-schedule, we were spared the end-of-project anguish that arises when it becomes clear that many tasks that were checked off weren’t actually done, and that countless other small issues weren’t adequately planned for. It was still a struggle to get the title completed on time, and we did work some long hours, but we never worked a seven-day week, and we were almost never at work after midnight. For us, that was a big accomplishment.

What Went Wrong

1. Started QA too late. We underestimated the scale and complexity of the QA effort that we needed to ship AMPLITUDE. FREQUENCY had shipped without a network mode (which we then developed subsequently), but we hadn’t needed to test both the core game and the networked game during the same development cycle; we had too few testers, and they started too late in the process.

Because the scope of our QA efforts was insufficient, there were a lot more



Design evolution from FREQUENCY's Activator to AMPLITUDE's Beatblaster.

bugs in AMPLITUDE than we realized. We were making final scheduling estimates based on a fairly modest number of bugs in our database (more than 200), under the mistaken assumption that those were all the bugs we had and that the title was in good shape. Once the QA department staffed up and got serious, our bug count ballooned and we had to scramble to hit our ship date. Lesson learned: QA for networked games is much more difficult and time-consuming than for non-networked games (especially on consoles, where games can't be patched).

2. Not enough art previsualization. We had scheduled some art previsualization time at the beginning of the project, but we ended up curtailing this phase pretty quickly and moving straight into production. This cost us time later in the project, as by the time production began, the art staff didn't have the look nailed down, and had to resort to time-consuming experimentation during the production cycle. The early press demos suffered as well, with less visual polish than we would have liked.

3. Too much to do, too little time. We had a clear mandate from our publisher to make our date; slippage was not an option. However,

we had pared our feature set to the minimum that we were comfortable shipping. We were very aware that as a sequel, AMPLITUDE had to constitute a significant advance beyond FREQUENCY to be accepted by the press and public. We were caught between a hard ship date and a feature set that we didn't think we could cut without potentially crippling the product in the marketplace. In practice, this meant that our carefully nurtured milestone-completion discipline began to unravel at the end of the project, and we ended up implementing core features well into beta. We didn't have nearly enough time to debug and tune a game of AMPLITUDE's complexity, and we ended up gambling that the competence and focus of the team and the solidity of our code base would see us through to a final candidate. While the gamble paid off, it was not a developmental approach that anyone would have liked to choose.

4. Fancy game shell more trouble than it was worth. The game shell was a very ambitious design, incorporating journeys through a fully realized 3D world as the player navigated between screens. The design was clearly more elaborate than was genuinely necessary and probably could have been scaled back early in development to retain most of its coolness without cost-

ing so much time and effort. However, we didn't provide sufficient managerial guidance as the early game shell environments were coming online, failing to rein in the scope of development until it was too late and we were locked into an implementation that was more elaborate and expensive than we really needed.

5. There is no number five. By and large, AMPLITUDE was an incredibly focused effort by an experienced team on a short cycle. We were very satisfied with how closely the final game met the developmental goals that were identified at the start of the project.

Conclusion

A MPLITUDE hit stores shelves in March 2003. So, were our theories correct? Did any of our changes help? AMPLITUDE's early sales are certainly outpacing FREQUENCY's, but it's still far too early to tell whether the title will become a hit. AMPLITUDE, like FREQUENCY, has been earning critical acclaim, and from our testing process, we have no doubt that if people give the game a try, they'll have trouble putting it down. For now we're waiting to find out whether the changes we made will help encourage a mass audience to give the game a shot. 🎮

An Indy Jones

There's an upcoming low-budget thriller called *The Deadly Percheron*, based on a bizarre cult novel about a psychiatrist, his deranged patient, and an imaginary leprechaun.

And it's being bankrolled by Rupert Murdoch and Fox News Corporation, who also brought us this summer's \$110 million sequel to *X-Men*.

Percheron was put into development by Fox Searchlight, Newscorp's "indy label," which distributes art house and international films. Searchlight didn't worry that the project was such odd, niche fare. Instead, says Howard Rodman, the veteran screenwriter who's adapting it, "They got that it was strange." They didn't mind, because it was marketable as a psychological thriller.

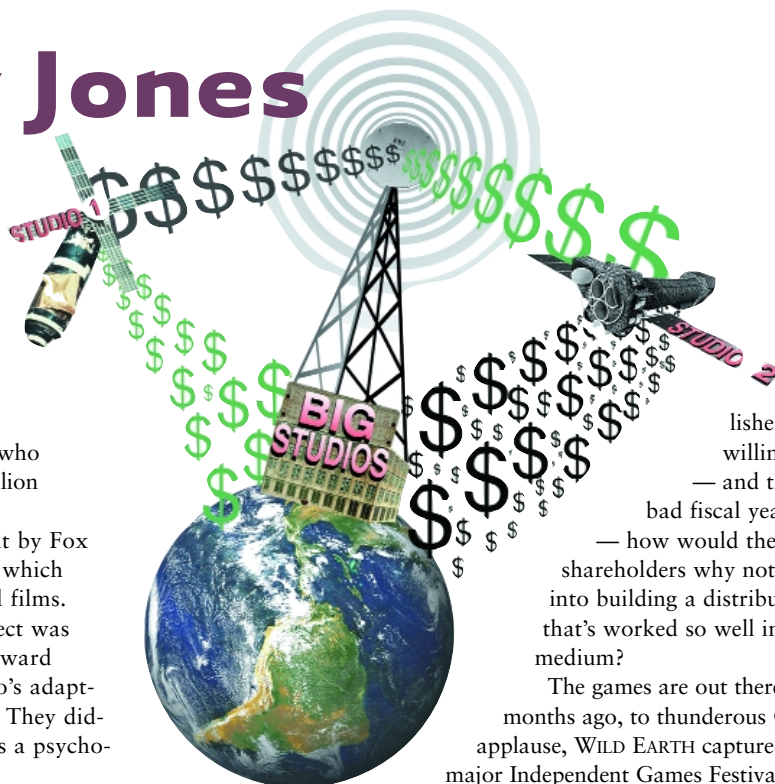
Most Hollywood studios have an indy label, and they don't finance them from any great love of art. It's really about "distribution of specialized product," as Rodman puts it; studios use their indy labels to find a market for films that wouldn't work in wide release but that, with savvy promotion, can become breakouts. The trick is to keep overhead, acquisition, and development costs down, so films such as *The Full Monty* or *Bend It Like Beckham* offset the inevitable bombs.

The side benefit in all this (and the thing most film-lovers appreciate) is that these indy labels are outlets for unique, groundbreaking films that drive the form forward.

By now, you've likely surmised where this is heading: Where are all the indy labels of the major game publishers?

Once more, with irony: Since many publishers are owned by media conglomerates that have indy labels in their film divisions, why don't they at least apply similar logic in their games? But there's no indy label at Fox Interactive, no Miramax at Disney Interactive, no Sony Pictures Classics at Sony Computer Entertainment, no Focus Features at Vivendi Universal Games.

The point here isn't to bash publishers. Rather, it's to point out a path that's already in their best interest to take: Why not invest a fraction of game profits to create an indy label and use it to distribute international and independently produced games which don't fit standard genre categories? And if pub-



lishers aren't willing to try this — and they have a bad fiscal year, anyway — how would they explain to shareholders why nothing went into building a distribution model that's worked so well in a related medium?

The games are out there. A few months ago, to thunderous GDC applause, *WILD EARTH* captured several major Independent Games Festival awards.

But according to head developer James Thrush, despite some promising leads (and a modest \$10,000 production budget), no publisher has yet put up the funds for its release. It looks like it eventually will be released, but even then, it would be among the few IGF winners to reach that pinnacle. Most major publishers, Thrush supposes, "are very short-sighted."

To be fair, publishers are understandably concerned about the risk. But surely there are ways to defray these fears, with the right angle of attack. By way of example, I offer two:

Small game, big name. Identifiable auteurs (for example Wes Anderson and Paul Thomas Anderson) are crucial to selling indy-label films. Similarly, there's potential in a game label, in which creative oversight over a few select indy games is provided by name developers. These games could be released under a rubric along the lines of "Renowned Developer Presents Obscure Indy Guy's Game." The developer's name and track record generates brand awareness (good for marketers), while increasing the chances for a left-field hit.

Don't believe something along those lines is feasible? I e-mailed Will Wright and asked him whether this was a prospect he'd be interested in. His quick reply: "I would say yes, assuming I had no contractual issues to consider (which I do), I think I would consider this idea." He added that Peter

continued on page 55

continued from page 56

Molyneux already does something similar with his Lionhead Studios' spin-off satellites such as Big Blue Box (FABLE).

Inside outsiders. When indie film labels succeed, says Rodman, it's largely attributable to the better executives running them, who have experience both in the studio system and in the rangy independent film scene. They bring a canny sense for fresh, unpredictable talent, and just as key, they know how to steward it toward commercial success. In the same way, publishers should bankroll business-minded developers who have a foot in both realms, and send them out onto the Internet, armed with a modest checkbook, in the hunt for wild game artistry.

Don't believe such inside outsiders exist? The judges roster at the IGF web site suggests otherwise.

From an economic standpoint, the

need to find new markets is pressing. In his invaluable weblog (May 28, 2003, entry), noted game designer Greg Costikyan predicts an imminent drought for the industry, with game sales tapering off, production costs accelerating, and audience growth flattening out. Ironically, Costikyan himself wrote a Soapbox in this space advocating for indie games ("A Platform for New Ideas," November 1999) — the passage of four years' time presents little progress and a far more dire outlook.

There are ways to make indie labels work, if we can only learn from the success stories in our sister media. In this, there's little to lose but a modest investment (which would have likely been burnt on derivative, underperforming titles anyway). There's a cascade of breakthroughs to gain.

So here is where the bravest voices on the business side of our industry must answer the call. (And developers should acclaim the efforts of the stars — let's give the IGDA's First Penguin Award to Eidos Interactive, for founding Fresh Games, a label bringing unique Japanese titles to the U.S. market.) Our thriving future depends on business visionaries to nobly count the necessary beans, and then to fling some out on the off-chance of magic happening. 🍀

WAGNER JAMES AU | James (wjamesau@well.com) is a contract designer/writer whose credits include MAJESTIC and SECOND LIFE, the object-building MMOG for which he writes "New World Notes" (www.seconddlife.com/notes). He also writes about games as an emerging art form for Salon.