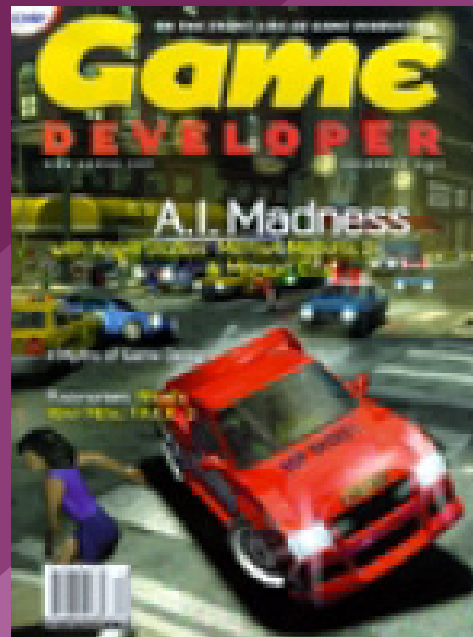




GAME DEVELOPER MAGAZINE

DECEMBER 2000



GAME PLAN

LETTER FROM THE EDITOR

Convergence

Imagine that you and your friends could go to the movie theater, and alongside your Hollywood favorites were additional movies that were interactive. These I-movies would be similar to movies in that they have a narrative structure, a high-resolution and high-fidelity presentation, and actors you know. But there the similarities end. To the side of the theater sits a guide, a person who plays the I-movie like a jazz musician plays an improvisation. The guide uses interactivity with the audience and his or her own personal whims to construct a linear narrative from the structure that is unique for each viewing. The participants get some control over the experience, like a game, and a satisfying linear narrative, like a movie.

It's been ten years since I first came across this concept, proposed by virtual reality pioneers such as Jaron Lanier. I've had it stuck in my mind ever since, and I subconsciously use it as a metric for the interactivity of films and games. What I'm talking about is a convergence of art forms, to create something new which has an interactivity level somewhere between zero (movies) and lots (games). The I-movie would be defined less rigorously in a narrative sense than a traditional movie or game, but the experience of it as played by the guide would be as linear as a movie. The guide would be someone who has been specially trained to deliver a compelling experience to the participants given the parameters laid out by the I-movie creator. Each I-movie would have a narrative direction and fundamental story line, but it would be open enough to allow for a unique experience each time.

New Technologies

We've all seen our fair share of movies with minimal (dorky) interactivity, and games that have so much FMV they're effectively interactive movies. We haven't found the sweet spot yet, but this convergence is heading for us at full steam.

Sony recently announced new technology which gives this concept a big powerup. They are in a unique position to promote real-time digital theater entertainment, having strong foundations in movies,

games, and music, as well as the coffer to support a hefty R&D investment. This year at Siggraph, Sony demonstrated the GScube, a box resembling a Borg Cube that's been fitted with 16 modified PlayStation 2 chipsets. This beast's theoretical performance is over one billion polygons per second. With this kind of power, we suddenly have the ability to create new real-time entertainment experiences.

There are some other great technologies being developed which also support this convergence. Vicon has a mo-cap system which SCEE's Soho Studios is using to do motion capture of four actors simultaneously. It can also be used for facial capture and the results synched with a voice recording. There are video cameras available that record color and distance from the camera; they are great for chromakey-style effects using real actors and rendered 3D environments. With wrap-around camera systems like those used in *The Matrix* (designed by Manex Visual Effects), it's possible to visually motion-capture a 3D sequence and convert it to real-time 3D for view-independent playback. It's truly an exciting time in the movie and game industries!

Welcome

This month marks a changing of the guard in our art column. Last month you experienced Mark Peasley's fabulous article on multi-legged animated critters. This month Maarten Kraaijvanger passes the art column over to Mark, who brings a wealth of experience with him from over ten years in the game industry. He has worked on such games as RED BARON, ACES OF THE PACIFIC, and (my personal favorite) STELLAR 7. Mark is currently the art director at Gas Powered Games. Find out more about Mark for yourself at his web site, www.pixelman.com. Welcome, Mark!



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

ON THE FRONT LINE OF GAME INNOVATION
Game DEVELOPER

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

Publisher
Jennifer Pahlka jpahlka@cmp.com

EDITORIAL

Editor-In-Chief
Mark DeLoura mdeloura@cmp.com

Senior Editor
Jennifer Olsen jolsen@cmp.com

Managing Editor
Laura Huber lhuber@cmp.com

Production Editor
R.D.T. Byrd tbyrd@cmp.com

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

Contributing Editors
Daniel Huebner dan@gamasutra.com
Jeff Lander jeff@darwin3d.com
Mark Peasley mpeasley@gaspowered.com

Advisory Board

Hal Barwood LucasArts
Noah Falstein The Inspiracy
Brian Hook Verant Interactive
Susan Lee-Merrow Lucas Learning
Mark Miller Group Process Consulting
Paul Steed Independent
Dan Teven Teven Consulting
Rob Wyatt The Groove Alliance

ADVERTISING SALES

Director of Sales & Marketing
Greg Kerwin gkerwin@cmp.com t: 415.947.6218

National Sales Manager
Jennifer Orvik jorvik@cmp.com t: 415.947.6217

Account Manager, Western Region, Silicon Valley & Asia
Mike Colligan mcolligan@cmp.com t: 415.947.6223

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

Account Manager, Eastern Region & Europe
Afton Thatcher athatcher@cmp.com t: 415.947.6224

Sales Representative/Recruitment
Morgan Browning mbrowning@cmp.com t: 415.947.6225

ADVERTISING PRODUCTION

Senior Vice President/Production Andrew A. Mickus
Advertising Production Coordinator Kevin Chanel

Reprints Stella Valdez t: 916.983.6971

CMP GAME MEDIA GROUP MARKETING

Senior MarCom Manager Jennifer McLean

Strategic Marketing Manager Darrielle Sadle

Marketing Coordinator Scott Lyon

Audience Development Coordinator Jessica Shultz

Sales Marketing Associate Jennifer Cereghetti



Game Developer is
BPA approved

CIRCULATION

Group Circulation Director Kathy Henry
Director of Audience Development Henry Fung

Circulation Manager Ron Escobar
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
Corporate President/COO John Russell

CFO John Day

Group President, Business Technology Group Adam K. Marder

Group President, Specialized Technologies Group Regina Starr Ridley

Group President, Channel Group Pam Watkins

Group President, Electronics Group Steve Weitzner

Senior Vice President, Human Resources Leah Landro

Senior Vice President, Global Sales & Marketing Bill Howard

Senior Vice President, Business Development Vittoria Borazio

General Counsel Sandra Grayson

Vice President, Creative Technologies Johanna Kleppe





Patent Holders Respond to Editorial

In Alex Dunne's September editorial ("Patents, Patterns, and Other Patter," Game Plan), he cited my patent as one that *Game Developer* readers should be aware of. While there's a respectable anti-patent position, usually it's heard from advocates of free software, not commercial game developers, who already license everything from characters to the right to run on a platform. We're willing to consider licensing our patents for free for 100 percent GPL'd free software projects. Commercial developers must license on commercial terms.

As for our physics technology being innovative, we're the first ones to make the hard cases work. We've been throwing high-poly jointed characters down circular staircases for years now and making it look right; others are still banging boxes around.

And as Jeff Lander and Chris Hecker's review showed (Product Review, September 2000) major packages are having trouble doing even that. Nor is this vaporware; we sell a plug-in for Softimage 3D and give away a free version, something MathEngine announced at Siggraph in 1999 but never shipped. It's easy to do bad physics; it's much harder to get it right.

We're in negotiations with various developers, but are not releasing details until the deals close. We expect this to be a significant technology for years to come, as platforms get faster and good physics becomes a standard part of games.

John Nagle
Animats
via e-mail

I just read your "Patents, Patterns, and Other Patter" editorial concerning GE Marching Cube software that appeared in the September 2000 issue of *Game Developer*. We appreciate your recognition of General Electric's work in this technology, and would like to take this opportunity to further clarify this matter.

You should know that GE has made other significant contributions in visualization technology as a result of its re-

search in medical diagnostic imaging and aerospace. The GE Marching Cubes algorithm described in U.S. Patent 4,710,876 is just one example. Some other examples of GE's contribution to this technological area are captured in U.S.

Patents 4,719,585; 4,729,098; 4,751,643; 4,791,567; 4,821,213; 4,879,668; 4,914,589; 4,984,157; 4,985,834; 5,166,876; 5,561,749; 5,542,036; and 5,590,248. They are also described in the book *The Visualization Toolkit: An Object-Oriented Approach to 3-D Graphics* (Prentice Hall, 1987), and in the article "Marching Cubes: A High Resolution 3D Surface Construction Algorithm" (ACM Computer Graphics, 1987), written by the inventors of this technology.

You should also know we have been granting licenses under these and other patents for various applications on fair and reasonable terms.

Thank you for taking the time to issue this clarification. We believe these additional comments will also be informative.

Jerald E. Roehling
GE Technology Development Inc.
via e-mail

Content More Worthy Than Form

Greg Costikyan's article on games and stories ("Where Stories End and Games Begin," September 2000) is, as always, insightful, analytical, and illuminating. After much effort, I was able to discern only two points to disagree with.

First, Greg holds that game and story are antithetical and must be traded off



against each other; I hold that story emerges from game. A game is a story-generator — a single playing of a game constitutes a story. Thus, I don't see game and story as antithetical; I see them as different levels of abstraction. A game is one level of abstraction higher than a story, just as "addition" is one level of abstraction higher than "three and two make five."

Second, Greg's arguments on art and emotion strike me as unnecessarily defensive. His overall thrust seems to be that games are every bit as good and worthy as stories or any other form of expression. I argue that the worthiness of any particular expression arises less from its form than its content. There are lots of unworthy novels, puerile paintings, and vulgar movies. There is nothing intrinsic to games as a form of expression that requires them to be unworthy, puerile, or vulgar. These adjectives must be applied to individual games or to particular groups of games, not to the medium in general and certainly not to its potential.

Thus, I find no contradiction in the claim that the content of most games is unworthy, puerile, or vulgar, while games as a form of expression could certainly be interpreted as noble, edifying, or worthy. Indeed, some of Greg's own games demonstrate the potential of the medium; would that they weren't swamped by the teeming masses of unworthy, puerile, and vulgar games.

Chris Crawford
via e-mail



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



FRONT LINE TOOLS

WHAT'S NEW IN THE WORLD OF GAME DEVELOPMENT | daniel huebner & mark deloura

3DMENOW DELIVERS LIP SERVICE

Biovirtual is adding lip-synching to its 3DMeNow package. 3DMeNow allows users to quickly create lifelike, animation-ready 3D models from source photographs by mapping the 2D photographs onto the features of a generic 3D template. The resulting models can be manipulated with 3DMeNow's 3D morphing, subdivision surface, and texture-blending tools and can be output as low-resolution game models or in a more highly detailed version. Biovirtual is extending the functionality of the package by using LIPSinc's technology to add accurate lip-synching with the input of any suitable .WAV file. 3DMeNow Pro is available for \$1,999.

3DMENOW | Biovirtual | www.biovirtual.com



Two examples of 3DMeNow creating ready-to-animate 3D models from photographs.

GUITAR RHYTHMS FROM MUSIC LAB



Rhythm'n'Chords delivers access to 700 guitar rhythm patterns.

more than 700 patterns. Pre-recorded patterns cover more than 60 accompaniment styles. Rhythm'n'Chords includes 22 guitar stroke types, such as down strums, up strums, muted strums, grace strums, slides, and plucking, with additional control parameters for strum velocity, balance, arpeggiation time, and polyphony. The plug-in includes a chord chart view as well as chord menu for the creation of chord progressions, and chord banks for storing chord configurations arranged by users. Rhythm'n'Chords for Cakewalk

MusicLab has released its guitar rhythms plug-in for Cakewalk Pro Audio. Rhythm'n'Chords comes with a guitar rhythm library of more

Pro Audio 9 for Windows 98 or NT is available from MusicLab for \$99. Rhythm'n'Chords | MusicLab | www.musiclab.com

NVIDIA'S MULTIMEDIA XBOX PROCESSOR

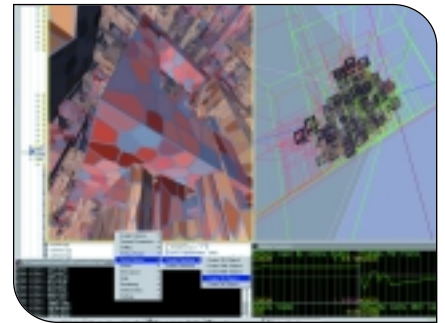
Nvidia has provided details about the second processor it is supplying for Microsoft's Xbox console. Dubbed the Xbox Media Communications Processor (MCPX), the part handles the broadband connectivity, communications, and audio capabilities of the Xbox. The MCPX includes dual DSPs, an audio processor, a Dolby Digital encoder, USB controller, modem interface, and an Ethernet controller to support home networking. Nvidia plans to begin selling an integrated chip late next year,

which will feature a modified version of the MCPX with additional memory controller functions.

MCPX | Nvidia | www.nvidia.com

HYBRID RELEASES VISIBILITY OPTIMIZER

Hybrid Holding has released SurRender Umbra, a visibility optimizer designed to identify visible objects in dynamic 3D environments as quickly as possible, without any scene pre-processing. Once Umbra has completed its tasks, the application can continue by drawing only the visible objects, leading to a savings in rendering time. The system works with environments of any topological structure and can be plugged into any game engine. The visibility queries are output-sensitive, meaning that SurRender Umbra's work time is dependent on the number of visible objects in the



SurRender Umbra identifies visible objects in dynamic 3D environments.

scene. Umbra is available for multiple platforms with end-product, royalty-free licenses running from \$10,000 for a single platform license with no support, to \$150,000 for all platforms with support. SurRender Umbra | Hybrid Holding Ltd. | www.hybrid.fi

NEW SOUNDBLASTERS LAUNCHED

Creative Technology has launched a new line of SoundBlaster cards, the Live! 5.1 series. This new series of sound cards supports 6-channel audio to deliver true Dolby Digital 5.1-channel surround sound via analog or digital connections. The Live! X-Gamer 5.1 and Live! MP3+ 5.1 both address specific gaming and music needs, while the Live! Platinum 5.1 offers consumers a high-end choice for digital audio creation and playback. The Live! Platinum 5.1 comes with the Live!

Drive IR, which allows simultaneous connection of multiple audio devices through a panel mounted in a drive bay. The Live! Platinum 5.1 also ships with a wireless remote control for easy navigation of audio and video playback utilities.

Creative Technology's X-Gamer 5.1 addresses specific gaming needs.

SOUNDBLASTER LIVE! 5.1 | Creative Technology | www.creative.com





Game restrictions. The Federal Trade Commission report on entertainment marketing is having a clear effect on the gaming industry. K-Mart and Wal-Mart are joining Toys R Us in restricting the sale of mature-rated games. K-Mart and Wal-Mart have announced plans to restrict sales of M-rated game titles to anyone under the age of 17, and a number of regional department stores are planning to follow suit. The stores will use a barcode scanner to remind clerks to check customer IDs.

While an Indianapolis ordinance governing the display and use of mature arcade games is being reviewed by the courts, a San Diego councilman is proposing a similar measure in that city. As in Indianapolis, games featuring violent or sexually explicit themes would need to be clearly marked and kept at least ten feet away from non-offensive games. Penalties include fines up to \$1,000 and revocation of game operator licenses. GameWorks, a chain of arcades, is introducing a new policy to restrict play of games identified as containing mature content. All 20 U.S. GameWorks and GameWorks Studio locations will only allow the sale of a limited-access debit card to patrons under the age of 16.

International debates are raging on the subject as well, though violence and mature subject matter are not in the spotlight. Officials in the southern Chinese city of Guangzhou have announced a plan to shut down more than 1,500 videogame arcades because of concerns about their influence on children. Parents and teachers in the region believe that the game parlors are distracting students from their studies and causing them to make friends with the wrong crowd. The crackdown affects almost 80 percent of Guangzhou's arcades, most of which are being cited for breaking the age restrictions. Many in the region, however, are calling for a total ban. In Malaysia, Home Minister Abdullah Ahmad Badawi has ordered that all arcades must close in two months. The primary motivation in that country centers on addiction and gambling as the reasons.

Learning Company details. Mattel won't see any immediate payment in its recent sale



DUKE NUKEM FOREVER. This game's probable M-rating would keep it from being sold at K-Mart and Wal-Mart stores.

of The Learning Company, which it acquired for \$3.5 billion in May 1999. Instead, the deal will allow Mattel to share in profits from future sales of The Learning Company's licensed products, and grants Mattel a chance to end the estimated \$60–90 million quarterly losses associated with The Learning Company. The buyer, Gores Technology Group, hasn't disclosed any immediate plans for The Learning Company, but has hinted that layoffs are likely on the way for some of the company's 1,500 employees.

Aureal buyout approved. Creative Technology has announced that its purchase of beleaguered audio hardware maker Aureal Semiconductor has been approved. The U.S. Bankruptcy Court for the Northern District of California, Oakland Division, entered the final order approving the sale of substantially all Aureal's assets to Creative, including patents, trademarks, and other intellectual property. The sale also includes settlement of all outstanding litigation claims between the two companies. Creative will pay \$28 million in cash, plus two new shares of Creative stock for every 100 outstanding shares of Aureal stock, which amounts to 208,079 shares of Creative, valued at approximately \$4.35 million based on the fair market value at the time of the sale.

Infogrames merger complete. Infogrames has finished consolidating its North American operations. The complex deal, involving nearly 50 million shares of Infogrames Inc. stock and the retirement

of \$128.6 million of debt, effectively combines the operations of the former GT Interactive with those of the Infogrames SA's North American subsidiary. The newly merged company will finally shed its outdated GTIS Nasdaq symbol in favor of IFGM. Infogrames also made changes in the management of its newly unified North American operations, adding Paradigm Entertainment's Dave Gatchel as senior vice president of development, and Cathy Tische from 3DO as vice president of licensing.

ATI posts loss. ATI Technologies has reported its second straight quarterly loss. The company saw revenues for the fourth quarter drop to \$290.2 million for a net loss of \$45.2 million. ATI posted earnings of \$16.8 million on revenues of \$304.7 million in the same quarter last year. After a difficult summer that included both the resignation of the company's chief financial officer and a third-quarter loss that triggered a stock slump, ATI is predicting 30 percent growth, quarter over quarter, for the first part of its 2001 fiscal year. 📈



UPCOMING EVENTS CALENDAR

MACWORLD EXPO

MOSCONE CONVENTION CENTER
San Francisco, Calif.
January 9–12, 2001
Cost: \$25 and up (early-bird discounts available)
www.macworldexpo.com

IDEA 2001 & GAME TECHNOLOGY CONFERENCE

HONG KONG CONVENTION AND EXHIBITION CENTRE
Hong Kong
Conference: January 17–21, 2001
Expo: January 18–21, 2001
Cost: (exhibits only) HK\$20 per day (conference) HK\$3,700
www.idea-expo.com



State Decision and Consequence Separation

a.k.a. Duplicated State Decision Points

submitted by dave weinstein

Introduction

Not only is this our first pattern to be selected from our readership, it is also our first “anti-pattern.” Not long after the release of the infamous *Design Patterns* book, another popular book, *AntiPatterns*, followed which enumerated patterns of software failure. Not surprisingly, it is often more valuable to know a problem than to know a solution.

Problem

Games frequently consist of large numbers of interrelated state transitions. The complexity of such state machines is difficult to manage, and this is especially true for multiplayer games where the likelihood of clients receiving messages inappropriate for their state must be addressed.

A common method of implementing such state transitions, usually because of lack of time or laziness, is to separate the decision-making process from the resulting consequences. For example, when a packet arrives, a flag is set in one module while several remote modules monitor that flag to modify their behavior.

This anti-pattern captures the complexity that results as such related code fragments increase. The greater the separation, both in terms of location in the source code and in logical association between decisions and consequences, and the more such code is part of ad hoc development rather than coherent design, the worse the anti-pattern gets.

Common problems include: difficult to follow code flow; state errors introduced as old functions are used as boilerplate for new code; the flow of state transitions is spread across the entire codebase leading to difficulty in changing or introducing new states; and code changes are increasingly vulnerable to human error because the distributed nature of the anti-pattern makes individual changes appear reasonable.

Examples

One common form of the anti-pattern is created by repeated state checks at the beginning of functions, aborting the function if the state is inappropriate. For example:

```
onMenuSelect() {
    if ( ! inMenuMode ) return;
}
```

This construct is initially a straightforward way to monitor state and may work well for small codebases containing a limited number of states. However, it becomes increasingly vulnerable to failure as numerous ad hoc decision points accumulate. This is especially true when other programmers integrate code and blindly copy poorly understood code fragments, propagating state decision points into places with unexpected and difficult to test results.

Solutions

Solutions involve minimizing disjoint decision and consequence processing. A common implementation combines the two operations (decision and consequence) into one module with related data structures or a class responsible for the entire state transition sequence. This should be bolstered with a well-documented process for how state transitions are to be added and how state-dependent code is to be integrated into this scheme. As an example, consider a multiplayer game which upon transitioning state also filters those packets which are irrelevant to the state. This might be done with an array of valid packets as in the following pseudocode:

```
changeState( newState ) {
    switch( newState )
    case mainMenuMode:
        validPacketsPtr =
            mainMenuModeValidPackets;
}
```

An alternative solution is to define a uniform interface that allows state-depend-

ent objects to be tested in a centralized manner. For example:

```
processMessage( Msg *msg ) {
    if ( msg->isValid() ) msg->process();
}
// ChangeEquipmentMsg
// inherits and extends Msg
ChangeEquipmentMsg::isValid() {
    return state == inventoryMode;
}
```

In this example, the is-message-valid rules are encoded into the message class. On the surface this seems to suffer the problems of the anti-pattern by distributing state decisions and consequences, but the code can be considered easier to follow because of the single point where message processing is aborted depending on the state checks.

Issues

It is crucial to document how to add and change states, and how to query them in your game to avoid ad hoc solutions like this anti-pattern describes. However, internal documentation is always the first thing to give when a schedule begins to slip, so this is a hard issue to overcome.

Related Patterns and References

Solutions to this pattern are related to the State and Strategy patterns. See *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* by William Brown and others (Wiley & Sons, 1998).

Credits

Thanks to Dave Weinstein from Red Storm Entertainment for submitting this anti-pattern! Be sure to follow up with Dave on www.gamasutra.com/patterns.

Send your patterns and idioms to us at patterns@d6.com.

PRODUCT REVIEW



THE SKINNY ON NEW TOOLS

Never Learn Another Editor!

MicroEdge's Visual SlickEdit 5.0

by mark deloura



There are so many programming editors out there: VI, Emacs, Notepad, Edit, Brief, Jed, Zeus, Epsilon, KEdit, CodeWright, FTE, GWD, Multi-Edit, Visual C++'s editor, and many others. With so many options available, what features differentiate them from each other?

If you're a Windows programmer, you've probably learned to get along with the Microsoft Visual C++ (VC) editor, so any replacement editor will have to get along with VC very well. On game console or handheld projects, you may not have an IDE or editor provided, instead you get to roll your own environment. For cross-platform projects, having one editor that works the same for each operating system and development environment would be fabulous. If you're a Unix hacker but you have to work under Windows, wouldn't it be great to have something with a bit more functionality than VI, perhaps something that has function prototype tooltips?

In this article I'll examine Visual SlickEdit 5.0 (henceforth known as Slick), a multi-platform editor which incorporates a great many programmer-friendly features. Because Slick has so many features, I'm going to concentrate on those parts which are particularly useful or innovative to you as a programmer.

Visual C++ Integration

Let's start at the top. Paramount for any editor is interoperability with VC, because VC is the dominant development environment for Windows. Most of you are familiar with the VC IDE and editor. How well does Slick cooperate?

In an ideal world you could use Slick in the VC editor pane. There are packages that do this, but I have yet to see any that

work properly. Slick runs as a separate application, and it has many hooks to facilitate cooperation with VC.

Most important is Slick's ability to dissect workspace (.DSW) and project (.DSP) files. As you can see in Figure 1, the Files tab on the left pane of Slick looks very similar to VC's. Unfortunately, Slick doesn't let you change or save workspace and project files for VC. This is perhaps the most annoying thing in Slick, so it's all uphill from here. But this means any time you want to add a file to your project you have to go back to VC and do it from there. Fortunately, Slick will detect the modification and auto-load the updated files. Projects which aren't VC (such as GCC projects) can be easily modified in Slick.

Building, debugging compiler errors, and executing your application all can be done from within Slick. Slick calls the VC command line utilities for each of these operations, and you stay within the editor. You can also configure the menu options to execute your own compilers and tools for custom projects. The regular expressions used for parsing the compiler errors can even be changed.

Slick doesn't do program debugging, profiling, or resource editing, so you'll still have to rely on VC or other packages for that functionality. Slick does integrate well with a variety of source code control systems, including SourceSafe, RCS, and PVCS.

Flexibility

The absolute best feature of Slick is its amazing flexibility. Everything is configurable. The primary configuration change you'll make is what editor should be emulated: CUA (standard Windows interface), Slick text edition, Brief, Epsilon,

VI, Emacs, VC, or ISPF (an OS/390 editor). I tested most of these and discovered them to be very useable emulations. Being a die-hard Unix hacker, I was most interested in the VI and Emacs modes. VI emulation worked very well and even emulated some of the more esoteric regular expression features. Emacs emulation was similarly well done, except many of the extended functions normally provided through Emacs Lisp were missing.

Extensive configurability is built into the core of Slick. Hotkeys are bound to macros (functions), and the macros can also be called up from the editor's command line. The macros are written in Slick-C, an interpreted C-like language. It's easy to modify existing macros or create new ones, bind them to hotkeys, and tie them to particular file types. You can also modify all of Slick's forms and dialog boxes. Slick includes a complete form editor, and you can edit any available form, or create new ones.

I'm completely stunned by the configuration capabilities of Slick. It's possible to tune the entire editor for any development kit, allowing you to use the editor as a "home base" to compile and execute from. This is truly an editor designed by programmers, for programmers.

Language Support

So how easily will Slick let you work with your code? Well, remember that this is an editor, not an IDE, though some-

MARK DELOURA | Mark is the editor-in-chief of Game Developer magazine. He has learned far too many editors and still falls back to VI when in a hurry. Harass him at the old Unix hackers' home via mdeloura@cmp.com.

- ★★★★★ excellent
- ★★★★ very good
- ★★★ average
- ★★ fair
- ★ don't bother

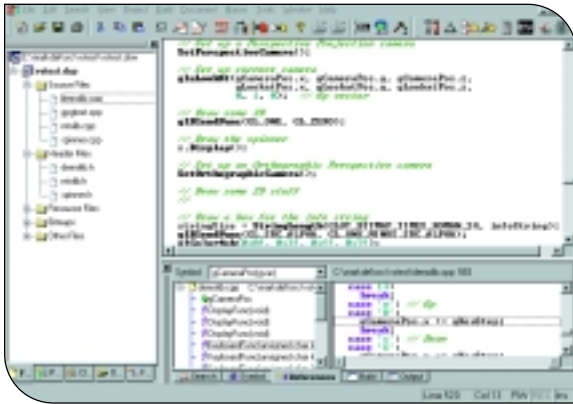


FIGURE 1. You can load Visual C++ workspace files into Visual SlickEdit. This is the Windows version, demonstrating the use of the References tab.

times that's easy to forget with so much functionality. You won't get to step through your code, peer at variables, or dump the register set. Slick does do a lot of things that you'd expect from an IDE though, and you'll be pleasantly surprised by many of the additional features that are included in this editor.

When you start Slick for the first time, it will prompt you for the location of your C/C++ header files and Java Development Kit in order to create tag files. Slick supports many languages, but particularly well supported are C/C++, Java/Javascript, HTML, and Slick-C. Of course, if your favorite language isn't supported, it's relatively simple to add your own macros to support syntax coloring and the like.

After Slick creates the tag files, you're ready to program. All my examples from here on will be regarding C++ support. Let's say you're working on a console title and need to link with the operating system libraries provided to you. The next thing you'll want to do is add all the OS code that you have available to the C tag file — from then on you'll get tooltips for all of your OS functions.

As you enter code, Slick parses it dynamically so that it can keep track of function prototypes, comment blocks, classes, and symbol references. If you create comment blocks above your functions, Slick automatically includes those comments in the function tooltips. You can also use Javadoc-style comments to include information for your functions. Using Javadoc you can add HTML-formatted comments to each function. This is amazingly useful: you can comment each parameter required for a function

call and include clickable links (for additional information), which pop up in the function tooltip.

As you enter code, the lower pane of the Slick interface seems to psychically interface with you in order to display useful information. If you have the Symbol tab selected, the lower pane will dynamically display information that Slick thinks you need to know. For example, type a class variable and the

file containing its class declaration will automatically open to help you select the proper member or function. If you have the References tab selected instead and hit Control-/, the lower pane will display all references to the symbol your cursor is on.

Features I Now Can't Live Without

A very useful and simple concept in Slick is "Selective Display." Selective Display modifies what you can see in the edit pane based on preprocessor definitions or function boundaries. I use this feature to collapse every function down to a simple "plus" box (like a Windows Explorer folder) with the function declaration. This does wonders for improving the readability of code. Just click the plus by the declaration to pop a function open, then edit it and click the plus again to close the function up.

Slick also performs code beautification. Set up your preferred tab spacing and curly bracket positions, then click "beautify" to apply those settings to all your existing code. All code that you type or paste in after beautification will also automatically adjust to your beautification settings and current tab level. This is especially useful when you have a team of people working on the same code.

Moving code between Windows and Unix is a breeze. Slick determines which OS the code came from based on line-ending character sequences, and then displays it without the nasty control sequences or merging the entire file into one long line. When saved, the code is then written out in its native format

regardless of the OS you're running on.

Some of the other very useful features in Slick include: an integrated FTP client which allows you to load-edit-save without ever saving to your local disk; a very full-featured differencing/merging utility (DIFFzilla); and a built-in hex editor.

Ballistics Report

The team at MicroEdge clearly understands what developers find useful in an editor. You can download a 30-day demo version of Slick from MicroEdge's web site.

I can't recommend it enough for its functionality as an editor. Using it under Linux, or with custom console compilers and linkers, is a dream. But when programming under Windows, working simultaneously with VC and Slick rapidly begins feeling like a chore. It would be nice to have just one tool, and even though the VC editor is inferior, it's often easier to just open VC for a quick edit and compile. *—*

VISUAL SLICKEDIT 5.0 ★★★★★

STATS

MicroEdge Inc.
Apex, N.C.
(919) 303-7400
www.slickedit.com

PRICE

\$295 for Windows or Linux versions; \$395 for other Unix versions.

SYSTEM REQUIREMENTS

Windows: 486DX or higher, 16MB RAM, 28MB hard disk
Linux: 486DX or higher, 24MB RAM, 40MB hard disk

PROS

1. Extensive configurability.
2. Huge feature set with innovative tooltip usage.
3. Great for Linux and console build environments.

CONS

1. Troublesome to use with Visual C++.
2. Hard to find what you're looking for in the GUI.
3. On the pricey side.

Pump Up The Volume

3D Objects That Don't Deflate



FIGURE 1. The crusher.

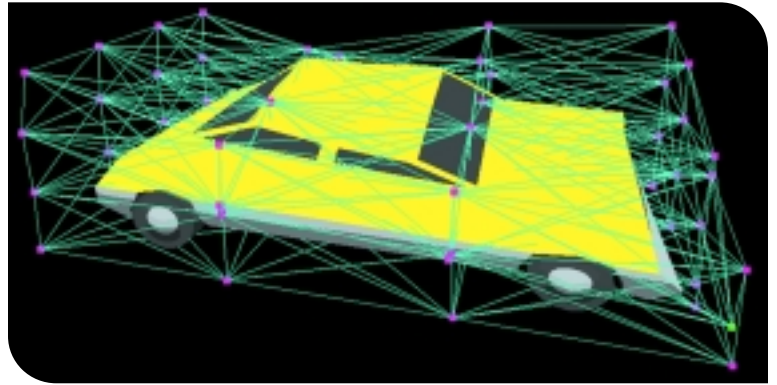


FIGURE 2. The crushed.

Like many kids who grew up in the suburbs, I spent the hours after school looking for something to keep me from having to confront my homework. The rule was, “Be home when the streetlights come on.” I would get home from school, chow down a quick mini-pizza (you gotta eat), then head out to basketball court to shoot hoops with the gang. This time of year was a bit of a drag. Growing up on the West Coast, I never needed to worry much about the cold in December, but the days were definitely getting shorter. We would just barely get a couple of games in before we had to pretend we could actually still see the basket. The streetlights would flicker on but that was easy enough to deny. It generally took a loud whistle from someone’s parent to break up the game.

I usually brought my own ball. If everyone brought a ball, we would never have a situation where we didn’t have one. Once everyone was there, the inspection began. Various qualities of faux leather were discussed. Overly worn or glassy-smooth balls were immediately discarded. The merits of over- and underinflating were then debated. We were all avid bike riders, so we had our pumps ready to correct any inflation issues.

Usually the problem was that no one could find one of those magic inflation needles. We could never find one of them when we needed it, even though I think I personally had bought dozens of them. Many nights we had to play with a “clunker” because no one had one of those damn needles. To this day I still treat those things with an odd kind of reverence. When I find one in the back of a drawer, I attempt to put it somewhere where I will immediately be able to find it when I need it. This inevitably means that I immediately lose it again. I am sure there are dozens of those things lying around here somewhere.

Why am I thinking about this? Well, some of my 3D models are looking a bit deflated lately, and I could really use one of those needles to pump them up.

Where’s Dig-Dug When You Need Him?

The image in Figure 1 is the cartoon taxi I created for this column in June (“In This Corner, the Crusher!”). In that column, I described the use of a dynamic mass-and-spring system connected to a matrix deformation lattice. This allowed me to make a polygonal mesh squash and

stretch like a cartoon object. The technique worked well, but there was a bit of a drawback to the system that I tried to pass off as a feature. The mesh system would occasionally collapse in on itself and stay that way, as you can see in Figure 2. For a car, that may be acceptable, but for many objects it would not work.

To solve the problem, I need to understand the reason this happens in the first place. My dynamics model is composed of a grid of point masses connected by springs. The springs initially attempt to keep each point the same distance away from each other. Let me take a cube as an example. The cube is composed of four point masses in each direction, making a total of 64 mass positions. Each of these is connected by a spring to each of its neighbors along the axes, as you can see in Figure 3. This gives me 27 smaller cube elements that make up the larger object.

When it’s just resting there, these springs are enough for the object to hold its shape. I call these structural springs. Unfortunately, those springs alone are not enough. I need to add springs across the diagonals to keep each small cube element from shearing or stretching too much. Those springs are enough to keep each small cube

JEFF LANDER | When the streetlights are not on, Jeff can be found playing with all sorts of high-tech toys at Darwin 3D. Drop him a note at jeff@darwin3d.com and make sure he gets home before it’s too late.

element from collapsing on itself. However, there is still a pretty big problem. Because the system is just a series of point masses connected by springs, it doesn't really represent a solid (but compressible) volume. Each element can pass through another one or occupy the same space with no penalty. As long as the connecting springs are at their rest length, all is well as far as the simulation is concerned. The image, however, looks plenty wrong, as you can see in Figure 4. The original cube is no longer discernible, even though the simulator has reached equilibrium.

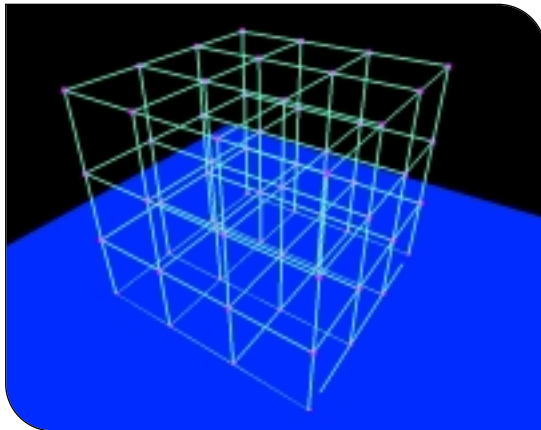


FIGURE 3. The spring cube.

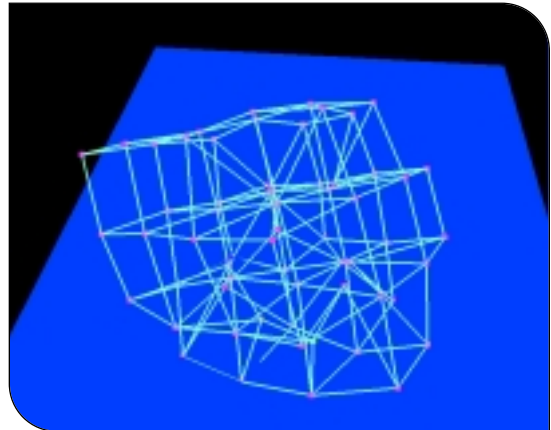


FIGURE 4. The collapsed cube.

Throw Me a Volume Preserver

Clearly, I need to make the simulator aware that each element contains a volume of material that would like to return to its rest position. Initially the impulse is simply to start adding more springs all over the place; connect every other mass node, then every third mass node, and so on. It should be obvious that this could rapidly degenerate into a situation where every mass node was connected to every other node by a spring. Extra spring calculations are a hit on performance, so it seemed that I should attack the problem from a different angle.

In mechanical engineering applications, the finite element method is often used for problems of deformation analysis. However, the method is fairly computationally expensive and difficult to implement for arbitrary models. For right now at least, I am not ready to give up on the mass-and-spring method. The goal for my deformable model is that it should deform naturally but tend to return to its initial shape when forces are not applied to counteract that tendency. The distance relationship between the mass nodes is initially given at load time. I really want those mass nodes not only to maintain equal distance between each other, but also

to maintain their locations relative to the initial local origin of the object. The problem with this is that because the object is free to move and tumble about in 3D space, I cannot simply pull the nodes back to the initial starting place. I need to create a local origin for the deformable object. For a rigid 3D body, this is easy. The object has both a center of mass (COM) and an orientation about that center. However, my soft-body object doesn't have a rigid COM or orientation, as that is constantly changing as the object deforms. I need to find a way to determine the center and orientation for the deformable object.

Journey to the Center of the Object

Once I determine the center of mass and orientation of a deformable body, it will be useful for a variety of things. Collision detection comes to mind right away. Determining whether two rigid bodies have collided is difficult. However, collision detection between two deformable bodies can become even more complicated. The bounding box of the object is not static. This provides me with a clue for how to determine the orientation of the object.

An oriented bounding box (OBB) is a box that surrounds an object in an orientation that provides a nice, tight fit around the object. I could use the orientation and center of an OBB for my local object axis. For an arbitrary set of 3D points, an OBB can be created by first finding the center of the box. The center of the box is determined by averaging the points in the

object. This average point is now considered the center of mass. That was the easy part; now it gets a bit trickier.

To determine the axes of the bounding box, I need to create a covariance matrix of the points in the model. The unit-length eigenvectors of this matrix are the axes of the OBB. If you are confused by terms like eigenvectors, you should pick up a linear algebra book. I never thought I would use it back when I was attending college, but more and more I find myself looking to linear algebra for all sorts of game applications. I've also found Dave Eberly's new book, *3D Game Engine Design* (see "For More Information"), very useful for calculating the OBB. His samples are very easy to adapt to situations like this. I'm not going to go into the OBB stuff this month because there was another problem once I had it working. (Isn't that always the way? Oh well, now I have the OBB generation code ready to go when I need it.)

The problem with the OBB code is while it returns a box that contains the object, the bounding box can be created in an arbitrary orientation as long as it contains the points of the object in a fairly efficient way. There is not necessarily any correspondence between the initial object orientation and the orientation of this bounding box. This is not a problem with the OBB algorithm. It works exactly as advertised; however, it caused me to refine my definition of what information I actually needed to solve the problem. I need an orientation with a correspondence to the initial object's fixed reference.

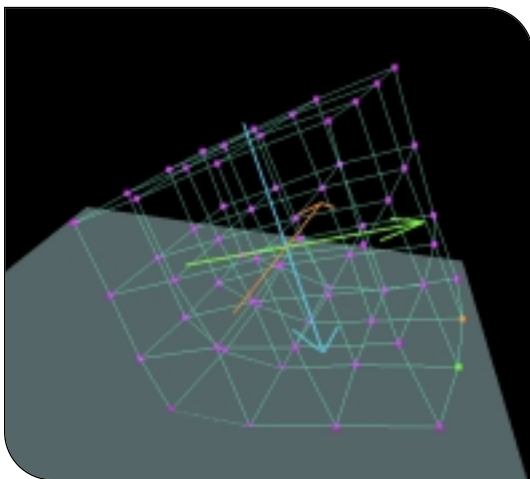
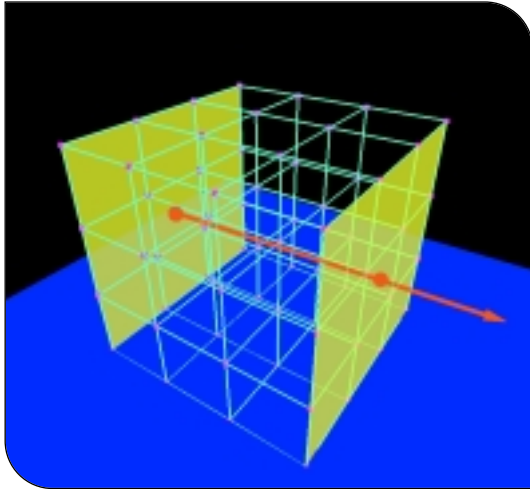


FIGURE 5 (top). Center vector found.

FIGURE 6 (bottom). The rotation axis in action.

Second Attempt at a Tumbling Axis

The next step was simply to consider the outer hull of the object and determine the principal axes from the orientation of the hull nodes. Let me first look at the left and right sides of the cube. I can calculate the average of the nodes on each side and create a vector between those two averaged points. This vector corresponds to the X-axis vector of the object's local coordinate frame. You can see this in Figure 5.

I can then do the same for the top and bottom nodes. This will give me the Y-axis vector for the object. To get the Z-axis, I can then just cross the X- and Y-axes using the cross product operator, giving me a Z-axis perpendicular to the two

other axes. Because the X- and Y-axes might not be perpendicular to each other in a deformable model, I need to do another cross product to make sure I have a valid rotational matrix. This matrix is representative of the deformed object in an arbitrary orientation. The code to calculate the transformation matrix for the deformed object can be found on the *Game Developer* web site. The center of mass position is stored in the translation portion of the matrix so the vertices can be easily transformed between coordinate frames. You can see the rotation axis in action in Figure 6.

Pump You Up

Now that I have a local transformation matrix for my deformable object, I can try to re-inflate my crushed cube. The initial mass node positions are transformed through the object matrix. This tells me where the nodes should be in the current orientation. I then use a spring for each node to pull the object back to its original shape. The strength of these springs is the strength with which the object regains its original shape. The springs should be strong enough

to keep the object from collapsing on itself, and can be used to simulate different types of materials which have different snap back properties.

Other Simulation Changes

While I was noodling in the dynamics simulation code, I changed a few things around from what I had been using. I changed the collision response system to use the penalty method instead of backing up the simulator to find the exact time of collision. When a collision is detected, a strong impulse is applied to the point of collision to force the objects apart. This system allows some penetration of the objects into the boundaries, but this is minimal. It also means that I don't need to

back up the simulator and search for the time of collision. I will discuss this system more next month.

I also changed the numerical integrator. If you remember my article on integration techniques ("Lone Game Developer Battles Physics Simulator," *Graphic Content*, April 1999), I had implemented several numerical integrators. I read about a variation on one in Jack Crenshaw's *Math Toolkit for Real-Time Programming* (see "For More Information") that I thought was interesting. If you recall from that column, the dynamics simulator uses an integrator to integrate the acceleration on a body for a timestep, h , to determine the velocity, and integrates the velocity for the timestep to determine the new position of the object. In that article, I treated both integration steps as separate operations. However, Crenshaw's book points out that dynamics simulations such as this use a second-order differential equation:

$$\frac{d^2 x}{dt^2} = f\left(t, x, \frac{dx}{dt}\right)$$

We reduce this to two first-order equations by introducing the variable, v , for velocity:


$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = f(t, x, v)$$

Crenshaw uses a predictor-corrector formula where the velocity is calculated using the previous acceleration and this generates the next step's velocity, which is used in the acceleration integration:

$$v_{i+1} = v_i + \frac{h}{2}(3f_i - f_{i-1})$$

$$x_{i+1} = x_i + \frac{h}{2}(v_{i+1} + v_i)$$

This integration technique seems to perform solidly and is very fast to calculate. Grab the source code and executable demo off the *Game Developer* web site at www.gdmag.com. 

FOR MORE INFORMATION

Crenshaw, Jack W. *Math Toolkit for Real-Time Programming*. Lawrence, Kans.: CMP Books, 2000.

Eberly, David H. *3D Game Engine Design*. San Francisco, Calif.: Morgan Kaufmann, 2000.

Unmasking Photoshop's Layers

Photoshop has been around for a long time, and has reigned supreme for most of that time. There is a good reason for that. In a word, it's "depth." Photoshop is one of those programs that seems to have more and more features and functionality the deeper you dig into it. Like peeling an onion, it reveals new and amazing things with each new layer that is exposed.

In this month's column, I'll explore some of the more advanced uses of layers and how a game artist might be able to take advantage of them in the everyday production environment. Some of the most powerful aspects of Photoshop are often not utilized by artists simply because they've not been exposed to where they might be useful in their everyday work. Let's take a familiar-sounding scenario and bring some advanced layering to bear on it in order to expose some of the flexibility.

Our story unfolds as our hero, Joe the artist, sits down in front of his computer for his morning coffee and e-mail routine. In rushes the designer/producer/art director (you choose) and says, "Sorry to dump this on you, Joe, but we need a logo for our new game idea, GUMBO'S REVENGE. I was thinking of a metallic, sort of old-world look and, you know, don't spend too much time on it, but make it look cool."

"Oh, and we need it by noon." (Insert big, heartfelt grin here.)

After a momentary bout of panic, Joe pops on his headphones and starts to build the ultimate layered Photoshop file (see example, above right). His goal is to create a document that is flexible, and can be altered easily without too much trouble. By using layer masks and clipping groups, he can create a file that is nondestructive. This means that it can be changed with little effort while maintaining its original look.

Here's how he did it:

First, an appropriate photo source for both the wood and the metal layer needs to be found



(see Figures 1 and 2). There is a ton of source material available from texture CDs, personal collections, or the web, but make sure that you know the copyrights associated with whatever file you choose. Another alternative is to use a program such as Bryce 4 or Corel Texture and make your own metal using some of the procedural textures applied to simple primitives. If you have access to a digital camera, a short walk around the building can glean a surprising number of high-quality textures.

We'll start out by creating a new file in Photoshop called GUMBO and then do a copy/paste of our source textures into the new file as layers. We will also add our logo elements on separate layers to make building our masterpiece easier.

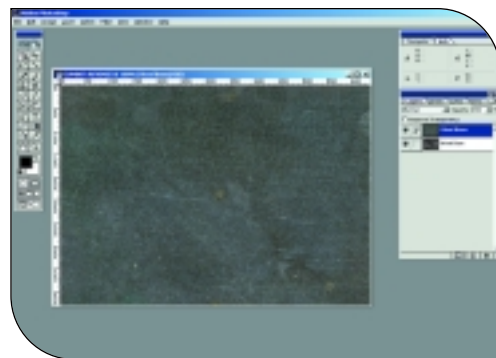
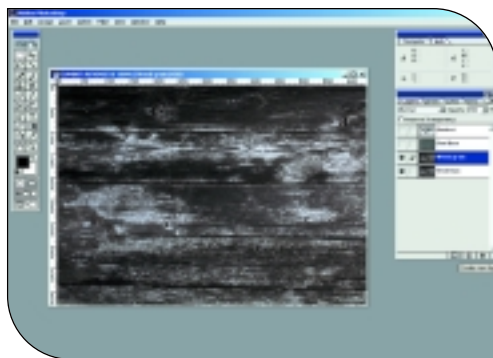


FIGURE 1 (above left) & FIGURE 2 (above right). Wood and metal photo sources used as a starting point.

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

Open up the wood source file and copy its contents into the clipboard. Paste this into a layer in your new GUMBO file and label it "Wood Base." It's a good idea to start out with labeling each new layer, as it will save time later. Next, copy the metal texture you have and paste it into the document as a new layer. Label it "Steel Base."

Now we'll add our speedily done logo to the mix (see Figure 3). In a vector program of choice, create a black and white version of the logo or graphic to be used. In this case, it was created in Coreldraw and exported as an Adobe Illustrator file. Over the years, I've found that it's generally easier to do all of my more advanced typographical manipulations in a vector-based program. They offer a much higher degree of control than can be found in the base Photoshop application. I highly recommend that game artists familiarize themselves with one of the main vector programs (such as Adobe Illustrator, Macromedia Freehand, or Coreldraw) and add them to your creative arsenal. A firm grasp of the basics and the ability to control the vector aspects of these programs can lead to untold hours of time saved in the raster arena.

I usually break any main graphic or logo elements out into individual files so that they can be imported into Photoshop on different layers. This is especially important if the various elements stack on top of one another in the design. In the case of our GUMBO logo, the elements don't overlap, so we can bring it in as one layer.

The files from the vector program need to be saved as either .AI (Adobe Illustrator) or .EPS (Encapsulated Postscript) files. Once in this format, they can be added as a new layer in Photoshop by using the Place command. The layer name is generated automatically based upon the vector file name. The advantage of using vector files and the Place function is that they are only rasterized when you have finally sized them to your desired width and height. This gives you the cleanest version of the rasterized logo, and it comes in with a transparent background. The color of the vector files is relatively unimportant, as we will be using these layers for selection sets for the most part.

Now that the base layers are in the file, we can begin to do some specialized work on the individual layers. First, let's look at our wood. It came in O.K., but we want to add some tooth or depth to the grain to make it pop a bit more.

In the Layers palette, Alt-click on the eyeball of the layer labeled Wood Base. This hides the other layers and makes only the Wood Base layer visible. Duplicate the wood layer (drag the layer down to the Create New Layer icon on the bottom of the palette window), and label the duplicate "Wood Grain." In order to see what is happening in the next step more easily, Alt-click on the eyeball of Wood Grain. This will make it the only visible layer (see Figure 4).

Now, double-click on the layer itself right over the label. This brings up the Layers Options window, which we need for the next step. Leave the default settings as they are and select the twin black pyramids below the grayscale bar labeled This Layer. Slide them to the right and notice what happens to your image. It begins to clip the black areas out of the image, leaving them transparent. Slide it to the right until about half of the dark area is transparent. Select the twin white pyramids to the right and begin to slide them

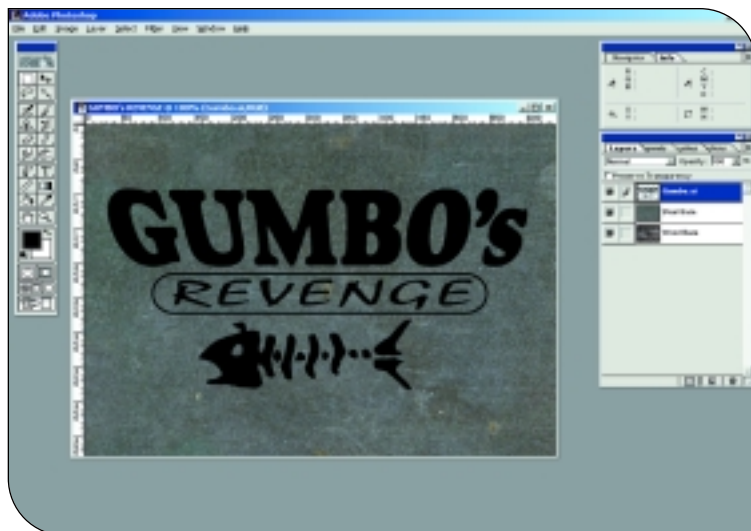
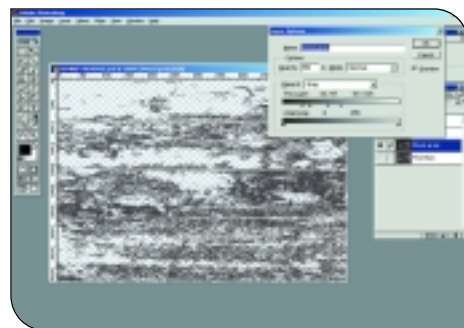
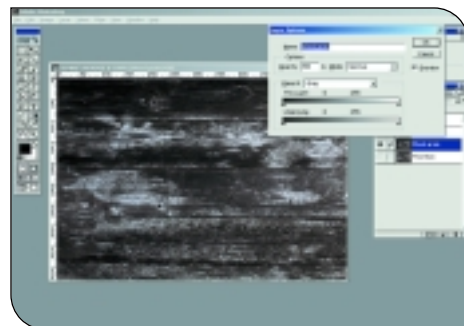


FIGURE 3 (above). The black and white version of the logo.

FIGURE 4 (right). The wood layer is duplicated and re-named Wood Grain.

FIGURE 5 (lower right). The layer is split and blended in the Layer Option panel.



towards the left. Eventually, you will see clipping of the white areas of the wood. Move the twin white pyramids to the left until you have dropped out about half of the remaining

white area. The clipping effect threshold for each range can be softened by pressing the Alt key while sliding the pyramids left or right. This splits each one into two parts and will give you a "soft" selection. Slightly soften each range (see Figure 5).

Now lock this transparency into your Wood Grain layer. Even though you see the effect on the screen, if you look at the layer's thumbnail, you will notice that it isn't displaying any transparency. Add a new layer between the Wood Base and Wood Grain layers, select the Wood Grain layer, and collapse it down to the empty layer (the shortcut for this is Control-E).

You should now see the new layer has the transparency we wanted locked down. A quick check of the layer icon will verify that it has the alpha applied. Unfortunately, when we collapsed it down the layer lost its name, so rename the new layer "Wood Grain" again.

Now for some fun — turn on the eyeball of both the Wood Grain and Wood Base layers. This makes the effect easier to see. Right-click on the Wood Grain layer and select Effects. Turn off

the Apply toggle on the Drop Shadow effect. In the pull-down menu, select Bevel and Emboss. Toggle the Apply on and start playing around with the various settings on the different pull-down menus. You will see some fairly dramatic effects that can be achieved easily (see Figure 6).

Another thing you can do is to add a layer mask to the Wood Grain layer. To do this, make sure the layer is selected, and then click on the lower-left icon on the Layers palette. This will add an alpha channel that is linked to the RGB layer, and flood-fill the alpha channel with white. The net result of this is no apparent change to the RGB image. However, what has happened is that an active, linked alpha channel has been created for that specific layer. To see what that means, select black as your foreground color, and select the paint tool of your choice. By painting with black on the layer mask, you are making that part of the image transparent. Conversely, by painting back in with white, the RGB portion of the image is brought back. The advantage of using a layer mask as opposed to just using an eraser on the RGB layer is that it is non-destructive. You can always go back into the layer mask and paint white back into the image, and it will become visible again.

Metal Effect

Now that you have the wood layer looking like you want, it's time to start working on the metal part of the logo. On our GUMBO logo, I've decided I wanted a base metal plate that the entire thing sits on, so I'll take the Steel Base layer and duplicate it. Rename the duplicate layer "Plate" and add a layer mask. Hold the Alt key down and click on the eyeball to make it the only visible active layer. Now, with the Control key depressed, click on the logo layer that was a result of the imported vector graphic. In the case of this file, it's called GUMBO.AI. When you do this you should see the hand cursor change to one with a small marquee added to it as you roll over the logo layer. This loads the transparency values of the layer as a selection set into the layer you are currently on.

Because we want to make this a plate that extends beyond our letters, we need to grow or expand the selection set. Go to the

Select>Modify>Expand menu available from the main menu bar. In the case of this logo, I chose 15 as the number of pixels to expand the current selection. Now, invert the selection by pressing Control-Alt-I. The last step is to fill black into the Plate layer mask with the selection we have. The result is a plate that surrounds where our logo will be (see Figure 7).

We can now add some layer effects to increase the visual interest of the plate. Right-click on the Plate layer in the Layers palette and select Effects. Bear in mind that the numbers are pixel-based. So if you are working on a high-resolution image, you will need to crank the numbers up quite a bit beyond what is being done on this 640x480 image. Also, click on the eyeball of both the Wood Base and Wood Grain layers. It makes it easier to see the results of the next step.

Leave the Drop Shadow toggle on, and set the distance to 5, the blur to 30, and the intensity to 100. Now, go to the main effect pull-down and select Bevel and Emboss. With this effect toggled on, set the style to Inner Bevel. Also set the depth to around 10 pixels and the blur to 5 pixels. Leave all other settings at their default values.

You should now see a plate of metal lying over the wood base. Depending upon the lettering style or logo, there may end up being some unwanted holes in your steel plate. This is easy to fix by going directly into the layer mask and painting with white to remove the holes. An easy way to see just the active layer mask is to Alt-click directly on the Plate layer mask thumbnail in the Layers palette. This will turn all other layers off and display only the layer mask. Alt-clicking again on the same thumbnail returns you back to where you started.

Adding the Logo

Now that we have the Plate layer in reasonable shape, let's add the logo and play around with how that's going to look. An easy way to get started is simply to drag the Plate layer down to the Create New Layer icon at the bottom of the Layers palette. This will create a duplicate layer with all effects and masks intact. Rename the copy "Letters" and fill its active layer

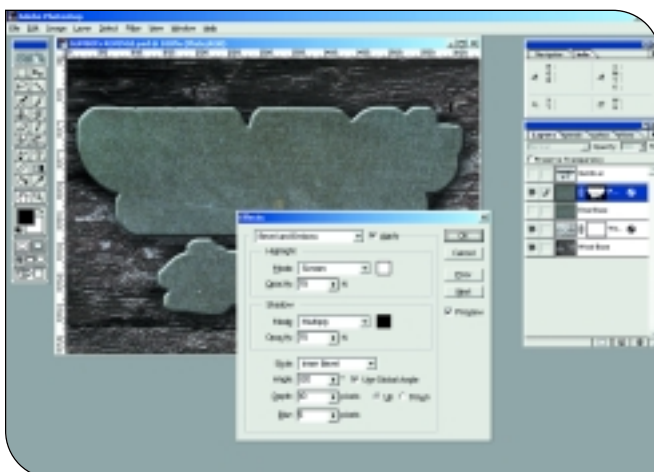


FIGURE 6. A Drop Shadow and a Bevel and Emboss effect are added to the layer. FIGURE 7. The Plate layer is duplicated and renamed "Letters."

mask with black. This will give you a layer that is visually empty, but has all of the settings we created for the Plate layer just waiting to be used. Make sure that the only visible layers are Letters, Plate, Wood Grain, and Wood Base.

Now load up the logo as a selection set. Do this by Control-clicking on the logo layer. Make sure that you have the Letters layer mask selected by clicking on the black thumbnail. It's sometimes easy to have the RGB channel selected instead of the layer mask. Visually, your only clue as to which one is active is a small black outline around the thumbnail itself. So, with the Letters layer mask selected, fill the selection set with white. You should immediately see a raised-letter version of the logo become visible. I readjusted the drop-shadow effect to a distance setting of 0 and a blur setting of 15 for better visual appeal.



FIGURE 8. The Adjustment layer is merged with the Bones layer for the final image.

Adding Another Plate Level

After a quick critique of the logo, I decided that an additional metal plate was needed between the word “Revenge” and the plate we just made. Once again, simply select and drag the Letters layer down to the Create New Layer icon and rename it “Plate 2.” Select the layer mask for Plate 2 and fill it with black. The newly created layer also needs to be placed in the correct order in the layers stack. To do this, select and drag the Plate 2 layer between the Plate and Letters layers.

Now we need to create a layer that will be used as a selection set for our new Plate 2 layer. I could always just load my logo layer up and do some fancy selecting and de-selecting, but if a mistake is made the penalty is starting over. An easier method is to build temporary layers that are used only for selection sets. These layers can be left invisible when not in use.

With that in mind, create a new layer and call it “Plate Selection.” In this layer, create a shape for the second plate and fill it with black. With the visibility of the Plate Selection layer turned off, make the Plate 2 layer active. Control-click on the Plate Selection layer to load it up as a selection set. Make sure the Plate 2 layer mask is selected (click on the thumbnail) and fill the selection with white. Just like that, we have another plate in the image that is fully editable.

Finishing Touches

As the noon hour draws near, Joe the artist puts on some finishing touches to give the image some pop. Once again, a non-destructive approach gives him the most flexibility with the file.

After some quick evaluation, Joe decides that the word “Gumbo” needs some color. He can take advantage of layer masking in another powerful way: clipping groups.

First, add a new layer above the Letters layer and call it “Gradient.” Make Gradient the active layer and fill it with a transparent ramp to red. Accomplish this by selecting the linear gradient tool, making red the foreground color, and setting the Options tab from Foreground to Transparent under the Gradient pull-down.

Once this is done, select Overlay from the layer blending mode pull-down at the top of the Layers palette. We now want to have this red gradient effect only visible in the word “Gumbo.” This can be accomplished easily by making the Gradient layer a clipping group of the Letters layer. Alt-click on the line between the Gradient layer and the Letters layer. When you have the Alt key depressed, the cursor will turn into two overlapping circles with an arrow when it rolls over the line. The successfully clipped layer can be identified by a dotted line between the two layers.

The next-to-last step is to make the fish bones look better. Duplicate the layer that contains the fish bones, which in our case was the Letters layer. Rename it “Bones,” and drag it to the top of the layers stack. Alt-click on the Bones layer mask to make it more clearly visible. Select everything except for the fish shape, and fill it in with black. Alt-click again to return the logo to the normal view.

Finally, add an Adjustment layer. Go to the main tool bar and select Layers>New>Adjustment Layer. Select Hue/Saturation from the available choices, and set the Saturation to -100 and the Lightness to +75. Make this a clipping group for the Bones layer by Alt-clicking on the line between the Hue/Saturation layer and the Bones layer (see Figure 8).

With only minutes to spare, our hero Joe has once again pulled through in a clinch and delivered a file that is not only flexible but easy to manipulate. This type of approach to building nondestructive graphics works extremely well when designing graphical user interfaces or any other elements in Photoshop that are prone to continual evolution and tweaking. The downside is that the file size becomes fairly large, but with today's systems, that is usually not an issue. By playing around with some of these advanced layers techniques, you'll find yourself with an amazing amount of control, and be able to increase your production speeds substantially. 🦄



Discuss this article in Gamasutra's Connection!

www.gamasutra.com/discuss/gdmag



Using AI to Bring Open City Racing to Life

Angel Studios' *MIDTOWN MADNESS 2* for PC and *MIDNIGHT CLUB* for Playstation 2 are open racing games in which players have complete freedom to drive where they please. Set in "living cities," these games feature interactive entities that include opponents, cops, traffic, and pedestrians. The role of artificial intelligence is to make the behaviors of these high-level entities convincing and immersive: opponents must be competitive but not insurmountable. Cops who spot you breaking the law must diligently try to slow you down or stop you. Vehicles composing ambient traffic must follow all traffic laws while responding to collisions and other unpredictable circumstances. And pedestrians must go about their routine business, until you swerve towards them and provoke them to run for their lives. This article provides a strategy for programmers who are trying to create AI for open city racing games, which is based on the success of Angel Studios' implementation of AI in *MIDTOWN MADNESS 2* and *MIDNIGHT CLUB*. The following discussion focuses on the autonomous architecture used by each high-level entity in these games. As gameplay progresses, this autonomy allows each entity to decide for itself how it's going to react to its immediate circumstances. This approach has the benefit of

creating lifelike behaviors along with some that were never intended, but add to gameplay in surprising ways.

AI Map: Roads, Intersections, and Open Areas

At the highest level, a city is divided into three primary components for the AI map: roads, intersections, and open areas (see Figure 1). Most of this AI map is composed of roads (line segments) that connect intersections. For our purposes, an intersection is defined as a 2D area in which various roads join. Shortcuts are just like roads, except they are overlaid on top of the three main component types. Shortcuts are used to help the opponents navigate through the various open areas, which by definition have no visible roads or intersections. Each of these physical objects is reflected in a software object.

The road object contains all the data representing a street, in terms of lists of 3D vertices. The main definition of the road includes the left/right boundary data, the road's centerline, and orientation vectors defined for each vertex in the definition. Other important road data includes the traffic lane definitions, the pedestrian sidewalk definition, road segment lengths, and lane width data. A minimum of four 3D vertices are used to define a road, and each list of ver-

JOE ADZIMA | Joe has been an AI programmer at Angel Studios for three years. During that time, he architected and implemented the entire AI system for *MIDTOWN MADNESS 1* and *2* for PC and *MIDNIGHT CLUB* for Playstation 2. Joe thanks Robert Bacon, Angel Studios' technical writer, for the exceptional editorial efforts Robert has applied to this article.

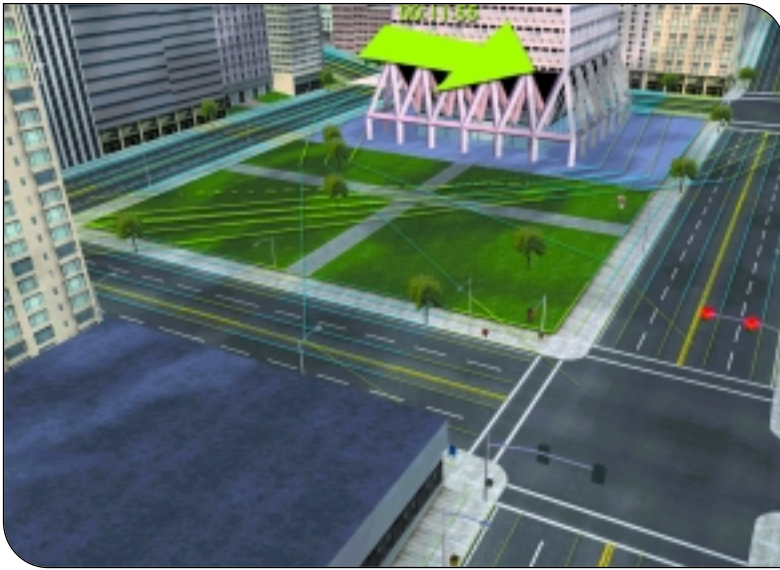


FIGURE 1. The AI map elements appear as green and blue line segments for roads and sidewalks, 2D areas for intersections, and additional line segments for shortcuts across open areas.

tices (for example, center vertices, boundary vertices, and so on) has the same number of vertices.

The intersection object contains a pointer to each connected shortcut and road segment. At initialization, these pointers are sorted in clockwise order. The sorting is necessary for helping the ambient traffic decide which is the correct road to turn onto when traversing an intersection. The intersection object also contains a pointer to a “traffic light set” object, which, as you might guess, is responsible for controlling the light’s sequence between green and red. Other important tasks for this object include obstacle management and stop-sign control.

Big-city solutions: leveraging the City Tool and GenBAI Tool. Angel’s method for creating extremely large cities uses a very sophisticated in-house tool called the City Tool. Not only does this tool create the physical representation of the city, but it also produces the raw data necessary for the AI to work. The City Tool allows the regeneration of the city database on a daily basis. Hence, the city can be customized very quickly to accommodate new gameplay elements that are discovered in prototyping, and to help resolve any issues that may emerge with the AI algorithms.

The GenBAI Tool is a separate tool that processes the raw data generated from the City Tool into the format that the run-time code needs. Other essential tasks that this GenBAI Tool performs include the creation of the ambient and pedestrian population bubbles and the correlation of cull rooms (discrete regions of the city) to the components of the road map.

Based on the available AI performance budget and the immense size of the cities, it’s impossible to simulate an entire city at once. The solution is to define a “bubble” that contains a list of all the road components on the city map that are visible from each cull room in the city, for the purpose of culling the simulation of traffic and pedestrians beyond a certain distance. This collection of road components essentially becomes the bubbles for ambient traffic and pedestrians.

The last function of the GenBAI tool is to create a binary version of the data that allows for superfast load times, because binary data can be directly mapped into the structures.

Data files: setting up races. The AI for each race event in the game is defined using one of two files: the city-based AI map data file or the race-based AI map data file. The city file contains

defaults to use for all the necessary AI settings at a city level. Each race event in the city includes a race-based AI map data file. This race file contains replacement values to use instead of the city values. This approach turns out to be a powerful design feature, because it allows the game designer to set defaults at a city level, and then easily override these values with new settings for each race.

Some examples of what is defined in these files are the number and definition of the race’s opponents, cops, and hook men. Also defined here are the models for the pedestrians and ambient vehicles to use for a specific race event. Finally, exceptions to the road data can be included to change the population fill density and speed limits.

Curves Ahead: Creating Traffic

Following rails and cubic spline curves. During normal driving conditions, all the ambient vehicles are positioned and oriented by a 2D spline curve. This curve defines the exact route the ambient traffic will drive in the XZ-plane. We used Hermite curves because the defining parameters, the start and end positions, and the directional vectors are easy to calculate and readily available.

Since the lanes for ambient vehicles on each road are defined by a list of vertices, a road subsegment can easily be created between each vertex in the list. When the ambient vehicle moves from one segment to the next, a new spline is calculated to define the path the vehicle will take. Splines are also used for creating recovery routes back to the main rail data. These recovery routes are necessary for recovering the path after a collision or a player-avoidance action sent the ambient vehicle off the rail. Using splines enables the ambient vehicles to drive smoothly through curves typically made up of many small road segments and intersections.

Setting the road velocity: the need for speed. Each road in the AI map has a speed-limit parameter for determining how fast ambient vehicles are allowed to drive on that road. In addition, each ambient vehicle has a random value for determining the amount it will drive over or under the road’s speed limit. This value can be negative or positive to allow the ambient vehicles to travel at different speeds relative to each other.

When a vehicle needs to accelerate, it uses a randomly selected value between 5 and 8 m/s². At other times, when an ambient vehicle needs to decelerate, perhaps because of a stop sign or red light, then the vehicle calculates a deceleration value based on attaining the desired speed in 1 second. The deceleration is calculated by

$$\frac{(V^2 - V_0^2)}{2(X - X_0)}$$

where V is the target velocity, V_0 is the current velocity, and $(X - X_0)$ is the distance required to perform the deceleration.

Detecting collisions. With performance times being so critical, each ambient vehicle can't test all the other ambient vehicles in its obstacle grid cell. As a compromise between speed and comprehensiveness, each ambient vehicle contains only a pointer to the next ambient vehicle directly in front of it in the same lane. On each frame, the ambient checks if the distance between itself and the next ambient vehicle is too close. If it is, the ambient in back will slow down to the speed of the ambient in front. Later, when the ambient in front becomes far enough away, the one in back will try to resume a different speed based on the current road's speed limit.

By itself, this simplification creates a problem with multi-car pile-ups. The problem can be solved by stopping the ambient vehicles at the intersections preceding the crash scene.

Crossing the intersection. Once an ambient vehicle reaches the end of a road, it must traverse an intersection. To do this, each vehicle needs to successfully gain approval from the following four functional groups.

First, the ambient vehicle must get approval from the intersection governing that road's "traffic control." Each road entering an intersection contains information that describes the traffic control for that road. Applicable control types are `NoStop`, `AllwaysStop`, `TrafficLight`, and `StopSign` (see Figure 2). If `NoStop` is set, then the ambient vehicle gets immediate approval to proceed through the intersection. If `AllwaysStop` is set, the ambient never gets approval to enter the intersection. If `TrafficLight` is set, the ambient is given approval whenever its direction has a green light. If `StopSign` is set, the ambient vehicle that has been waiting the longest time is approved to traverse the intersection.

The second approval group is the accident manager. The accident manager keeps track of all the ambient vehicles in the intersection and the next upcoming road segment. If there are any accidents present in these AI map components, then approval to traverse the intersection is denied. Otherwise, the ambient vehicle is approved and moves on to the third stage.

The third stage requires that the road which the ambient is going to be on after traversing the intersection has the road capacity to accept the ambient vehicle's entire length, with no part of the vehicle sticking into the intersection.

The fourth and final approval comes from a check to see if there are any other ambient vehicles trying to cross at the same time. An example of why this check is necessary is when an ambient vehicle is turning from a road controlled by a stop sign onto a main road controlled by a traffic light. Since the approval of the stop sign is based on the wait time at the intersection, the vehicle that's been waiting longest would have permission to cross the intersection — but in reality that vehicle needs to wait until the cars that have been given permission by the traffic light get out of the way.

Selecting the next road. When an ambient vehicle reaches the end of the intersection, the next decision the vehicle must make is which direction to take. Depending on its current lane assignment, the ambient vehicle selects the next road based on the following rules (see Figure 2):

- If a vehicle is in the far-left lane, it can go either left or straight.
- If it's in the far-right lane, it can go either right or straight.
- If it's in any of the center lanes, then it must go straight.



FIGURE 2. In this case, the `TrafficLight` class is set to red for some vehicles, which stop and wait. The other vehicles with green/yellow lights get permission to cross the intersection. The vehicle crossing in the left lane decides to turn left, while the vehicle in the right lane goes straight.

- If it's on a one-way road, then it picks randomly from any of the outgoing roads.
- If it's on a freeway intersection where on-ramps merge with the main freeway traffic, then it must always go right.
- U-turns are never allowed, mostly because a splined curve in this situation would not look natural.

Since the roads are sorted in clockwise order, this simplifies selection of the correct road. For example, to select the road to the left, just add 1 to the current road's intersection index value (the ID number of that road in the intersection road array). To pick the straight road, add 2. To go right, just subtract 1 from the road's intersection index value.

Changing lanes. On roads that are long enough, the ambient vehicles will change lanes in order to load an equal number of vehicles into each lane of the road. When the vehicle has traveled to the point that triggers the lane change (usually set at 25 percent of the total road length), the vehicle will calculate a spline that will take it smoothly from its current lane to the destination lane.

The difficulty here is in setting the next-vehicle pointer for collision detection. The solution is to have a next-vehicle pointer for each possible lane of the road. During this state, the vehicle is assigned to two separate lanes and therefore is actually able to detect collision for both traffic lanes.

Once a vehicle completes the lane change, it makes another decision as to which road it wants to turn onto after traversing the upcoming intersection. This decision is necessary because the vehicle is in a new lane and may not be able to get to the previously selected road from its new lane assignment.

Orienting the car. As the ambient traffic vehicles drive around the city, they are constantly driving over an arbitrary set of polygons forming the roads and intersections. One of the challenges for the AI is orienting the ambient vehicles to match the contour of the road and surfaces of open areas. Because there are hills, banked road surfaces, curbs separating roads and sidewalks, and uneven open terrain, the obvious way to orient the vehicles is to shoot a probe straight down the Y-axis from the front-left, front-right, and rear-left corners of the vehicle. First, get the XZ position of the vehicle from the calculated spline position and determine the three corner positions in respect to the center point of the vehicle. Then,

shoot probes at the three corners to get their Y positions.

Once you know the three corner positions, you can calculate the car's orientation vectors. This approach works very well, but even caching the last polygon isn't fast enough to do all the time for every car in the traffic bubble. One way to enhance performance is to mark every road as being either flat or not. If an ambient vehicle drives on a flat road, it doesn't need to do the full probe method. Instead, this vehicle could use just the Y value from the road's rail data. Another performance enhancement is to orient the vehicles that are far enough from the player using only the road's rail-orientation vectors. This approach works well when small vehicle-orientation pops are not noticeable.

Managing the collision state. When an ambient vehicle collides with the player, or with a dynamic or static obstacle in the city, the ambient vehicle switches from using a partially simulated physics model to a fully simulated physics model. The fully simulated model allows the ambient vehicle to act correctly in collisions.

A vehicle manager controls the activities of all the vehicles transitioning between physics models. A collision manager handles the collision itself. For example, once a vehicle has come to rest, the vehicle manager resets it back to the partially simulated physics model. At this point, the ambient vehicle attempts to plot a spline back to the road rail. As it proceeds along the rail, the vehicle will not perform any obstacle detection, and will collide with anything in its way. A collision then sends the vehicle back to the collision manager. This loop will repeat for a definable number of tries. If the maximum number of tries is reached, the ambient vehicle gives up and remains in its current location until the population manager places it back into the active bubble of the ambient vehicle pool.

Using an obstacle-avoidance grid. Every AI entity in the game is assigned to a cell in the obstacle-avoidance grid. This assignment allows fully simulated physics vehicles to perform faster obstacle avoidance.

Since the road is defined by a list of vertices, these vertices make natural separation points between obstacle-avoidance buckets. Together, these buckets divide the city into a grid that limits the scope of collision detection. As an ambient vehicle moves along its rail, crossing a boundary between buckets causes the vehicle to be removed from the previous bucket and added to the new bucket. The intersection is also considered an obstacle bucket.

Simulation bubbles for ambient traffic. A run-time parameter specifies the total number of ambient vehicles to create in the city. After being created, each ambient vehicle is placed into an ambient pool from which the ambients around the player are populated. This fully simulated region around the player is the simulation bubble. Relative to the locations of the player, remote regions of the city are outside of the simulation bubble, and are not fully simulated.

When a player moves from one cull room to another, the population manager compares the vertex list of the new cull room against the list for the old one. From these two lists, three new lists are created: New Roads, Obsolete Roads, and No Change Roads. First, the obsolete roads are removed from the active road list, and the ambient vehicles on them are placed into the ambient pool. Next, the new roads are populated with a vehicle density equal to the total vehicle length divided by the total road length. The vehicle density value is set to the default value based on the road type, or an

exception value set through the definition of the race AI map file.

As the ambient vehicles randomly drive around the city, they sometimes come to the edge of the simulation bubble. When this happens, the ambient vehicles have two choices. First, if the road type is two-way (that is, ambient vehicles can drive in both directions), then the vehicle is repositioned at the beginning of the current road's opposite direction. Alternatively, if the ambient vehicle reaches the end of a one-way road, the vehicle is removed from the road and placed into the pool and thereby becomes available to populate other bubbles.

Driving in London: left becomes right. London drivers use the left side of the road instead of the right. To accommodate this situation, some changes have to be made to the raw road data. First, all of the right lane data must be copied to the left lane data, and vice versa. The order of each lane's vertex data must then be reversed so that the first vertex becomes the last, and the lane order reversed so that what was the lane closest to the road's centerline becomes the lane farthest from the center.

Given these changes, the rest of the AI entities and the ambient vehicle logic will work the same regardless of which side of the road the traffic drives on. This architecture gave us the flexibility to allow left- or right-side driving in any city.

City People: Simulating Pedestrians

In real cities, pedestrians are on nearly every street corner. They walk and go about their business, so it should be no different in the cities we create in our games. The pedestrians wander along the sidewalks and sometimes cross streets. They avoid static obstacles such as mailboxes, streetlights, and parking meters, and also dynamic obstacles such as other pedestrians and the vehicles controlled by the players. And no, players can't run over the pedestrians, or get points for trying! Even so, interacting with these "peds" makes the player's experience as a city driver much more realistic and immersive.

Simulation bubbles for pedestrians. Just as the ambient traffic has a simulation bubble, so do the pedestrians. And while the pedestrian bubble has a much smaller radius, both types are handled similarly. During initialization, the pedestrians are created and inserted into the pedestrian pool. When the player is inserted into the city, the pedestrians are populated around him. During population, one pedestrian is added to each road in the bubble, round-robin style, until all the pedestrians in the pool are exhausted.

Pedestrians are initialized with a random road distance and side distance based on an offset to the center of the sidewalk. They are also assigned a direction in which to travel and a side of the street on which to start. As the pedestrians get to the edge of the population bubble, they simply turn around and walk back in the opposite direction from which they came.

Wandering the city. When walking the streets, the pedestrians use splines to smooth out the angles created by the road subsegments. All the spline calculations are done in 2D to increase the performance of the pedestrians. The Y value for the splines is calculated by probing the polygon the pedestrian is walking on in order to give the appearance that the pedestrian is actually walking on the terrain underneath its feet.



FIGURE 3 (above). In this situation, the *PreCrossStreet* state has moved the pedestrians next to the street curb, and now the *WaitToCrossStreet* state is holding the peds in place until the light turns green. FIGURE 4 (top right). When the oncoming player vehicle threatens these pedestrians, they decide to hug the wall after sending out a probe and finding a wall nearby. FIGURE 5 (bottom right). The pink lines visualize the direction the peds intend to walk. When a player vehicle introduces a threat, the pedestrians decide to dive right or left at the last moment, since no wall is nearby.

Each pedestrian has a target point for it to head toward. This target point is calculated by solving for the location on the spline path three meters ahead of the pedestrian. In walking, the ped will turn toward the target point a little bit each frame, while moving forward and sideways at a rate based on the parameters that control the animation speed. As the pedestrian walks down the road, the ped object calculates a new spline every time it passes a side-walk vertex.

Crossing the street. When a pedestrian gets to the end of the street, it has a decision to make. The ped either follows the sidewalk to the next street or crosses the street. If the ped decides to cross the street, then it must decide which street to cross: the current or the next. Four states control ped navigation on the streets: *Wander*, *PreCrossStreet*, *WaitToCrossStreet*, and *CrossStreet* (see Figure 3). The first of these, *Wander*, is described in the previous section, “Wandering the City.” *PreCrossStreet* takes the pedestrian from the end of the street to a position closer to the street curb, *WaitToCrossStreet* tells the pedestrian waiting for the traffic light that it’s time to cross the street, and *CrossStreet* handles the actual walking or running of the pedestrian to the curb on the other side of the street.

Animating actions. The core animation system for the pedestrians is skeleton-based. Specifically, animations are created in 3D Studio Max at 30FPS, and then downloaded using Angel’s proprietary exporter. The animation system accounts for the nonconstant nature of the frame rate.

For each type of pedestrian model, a data file identifies the animation sequences. Since all the translation information is removed from the animations, the data file also specifies the amount of translation necessary in the forward and sideways directions. To move the pedestrian, the ped object simply adds the total distance multiplied by the frame time for both the forward and sideways directions. (Most animation sequences have zero side-to-side movement.)

Two functions of the animation system are particularly useful. The *Start* function immediately starts the animation sequence specified as a parameter to the function, and the *Schedule* function

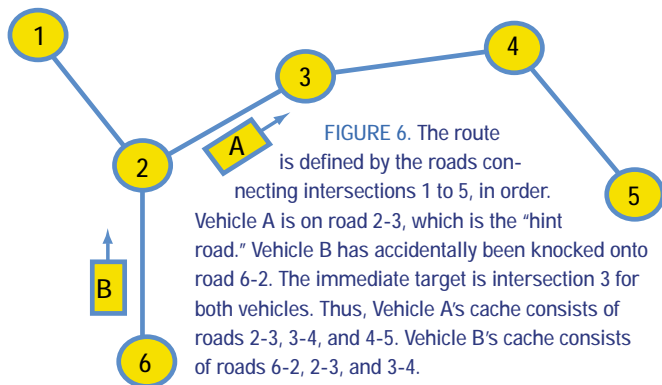


starts the desired animation sequence as soon as the current sequence finishes.

Avoiding the speeding player. The main rule for the pedestrians is to always avoid being hit. We accomplish this in two ways. First, if the pedestrian is near a wall, then the ped runs to the wall, puts its back against it, and stands flush up against it until the threatening vehicle moves away (see Figure 4). Alternatively, if no wall is nearby, the ped turns to face the oncoming vehicle, waits until the vehicle is close enough, and then dives to the left or right at the very last moment (see Figure 5).

The pedestrian object determines that an oncoming vehicle is a threat by taking the forward directional vector of the vehicle and performing a dot product with the vector defined by the ped’s position minus the vehicle’s position. This calculation measures the side distance. If the side distance is less than half the width of the vehicle, then a collision is imminent.

The next calculation is the time it will take the approaching vehicle to collide with the pedestrian. In this context, two distance zones are defined: a far and a near. In the far zone, the pedestrian turns to face the vehicle and then goes into an “anticipate” behavior, which results in a choice between shaking with fear and running away. The near zone activates the “avoid” behavior, which causes the pedestrian to look for a wall to hug. To locate a wall, the pedestrian object shoots a probe perpendicular to the sidewalk for ten meters from its current location. If a wall is found, the pedestrian runs to it. Otherwise, the ped dives in the opposite direction of the vehicle’s rotational momentum. (Sometimes the vehicle is going so fast, a superhuman boost in dive speed is needed to avoid a collision.)



Avoiding obstacles. As the pedestrians walk blissfully down the street, they come to obstacles in the road. The obstacles fall into one of three categories: other wandering pedestrians; props such as trash cans, mailboxes, and streetlights; or the player’s vehicle parked on the sidewalk.

In order to avoid other pedestrians, each ped checks all the pedestrians inside its obstacle grid cell. To detect a collision among this group, the ped performs a couple of calculations. First, it determines the side distance from the centerline of the sidewalk to itself and the other pedestrian. The ped’s radius is then added to and subtracted from this distance. A collision is imminent if there is any overlap between the two pedestrians.

In order to help them avoid each other, one of the pedestrians can stop while the other one passes. One way to do this is to make the pedestrian with the lowest identification number stop, and the latter ped sets its target point far enough to left or right to miss the former ped. The ped will always choose left if it’s within the sidewalk boundary; otherwise it will go to the right. If the right target point is also past the edge of the sidewalk, then the pedestrian will turn around and continue on its way. Similar calculations to pedestrian detection and avoidance are performed to detect and avoid the props and the player’s vehicle.

Simulating Vehicles with Full Physics

The full physics simulation object, `VehiclePhysics`, is a base class with the logic for navigating the city. The different entities in the city are derived from this base class, including the `RouteRacer` object (some of the opponents) and the `PoliceOfficer` object (cops). These child classes supply the additional logic necessary for performing higher-level behaviors. We use the term “full-physics vehicles” because the car being controlled for this category behaves within the laws of physics. These cars have code for simulating the engine, transmission, and wheels, and are controlled by setting values for steering, brake, and throttle. Additionally, the `VehiclePhysics` class contains two key public methods, `RegisterRoute` and `DriveRoute`.

Registering a route. The first thing that the navigation algorithm needs is a route. The route can either be created dynamically in real time or defined in a file as a list of intersection IDs. The real-time method always returns the shortest route. The file method is created by the Race Editor, another proprietary in-house tool that allows the game designer to look down on the city in 2D and select the intersections that make up the route. The game designer can thereby create very specific routes for opponents. Also, the file method eliminates the need for some of the AI entities to calculate

their routes in real time, which in turn saves processing time.

Planning the route. Once a route to a final destination has been specified, a little bit more detailed planning is needed for handling immediate situations. We used a road cache for this purpose, which stores the most immediate three roads the vehicle is on or needs to drive down next (see Figure 6).

At any given moment, the vehicle knows the next intersection it is trying to get to (the immediate target), so the vehicle can identify the road connecting this target intersection with the intersection immediately before the target. If the vehicle is already on this “hint road,” then the cache is filled with the hint road and the next two roads in the route.

If the vehicle isn’t on the hint road, it has gotten knocked off course. In this situation, the vehicle looks at all the roads that connect with the intersection immediately before the target. If the vehicle is on one of these roads, then the cache is filled with this road and the next two roads the vehicle needs to take in order to get back on track. If the vehicle isn’t on any of these roads, then it dynamically plots a new route to the target intersection.

Determining multiple routes. If there are no ambient vehicles in the city, then there is only one route necessary to give to an opponent (the computer-controlled player, or CCP), the best route. In general, however, there is ambient traffic everywhere that must be avoided if the CCP is to remain competitive. The choice then becomes which path to pick to avoid the obstacles. At any given moment, this choice comes down to going left or right to avoid an upcoming obstacle. As the CCP plans ahead, it determines two additional routes for each and every obstacle, until it reaches the required planning distance. This process produces a tree of routes to choose from (see Figure 7).

Choosing the best route. When all the possible routes have been enumerated, the best route for the CCP can be determined. Sometimes one or more of the routes will take the vehicle onto the sidewalk. Taking the sidewalk is a negative, so these routes are less attractive than those which stay on the road. Also, some routes will become completely blocked, with no way around the obstacles present, making those less attractive as well. The last criterion is minimizing the amount of turning required to drive a path. Taking all these criteria into account, the best route is usually the one that isn’t blocked, stays on the road, and goes as straight as possible.

Setting the steering. The CCP vehicle simulated with full physics uses the same driving model that the player's vehicle uses. For example, both vehicles take a steering parameter between -1.0 and 1.0. This parameter is input from the control pad for the player's vehicle, but the CCP must calculate its steering parameter in real time to avoid obstacles and reach its final destination. Rather than planning its entire route in advance, the CCP simplifies the problem by calculating a series of Steering Target Points (STPs), one per frame in real time as gameplay progresses. Each STP is simply the next point the CCP needs to steer towards to get one frame closer to its final destination. Each point is calculated with due consideration to navigating the road, navigating sharp turns, and avoiding obstacles.

Setting the throttle. Most of the time a CCP wants to go as fast as possible. There are two exceptions to this rule: traversing sharp turns and reaching the end of a race. Sharp turns are defined as those in which the angle between two road segments is greater than 45 degrees, and can occur anywhere along the road or when traversing an intersection. Since the route through a sharp turn is circular, it is easy to calculate the maximum velocity through the turn by the formula

$$V = \sqrt{ugR}$$

where V is equal to the velocity, u is the coefficient of friction for the road surface, g is the value of gravity, and R is the radius of our turn. Once the velocity is known, all that the CCP has to do

is slow down to the correct speed before entering the turn.

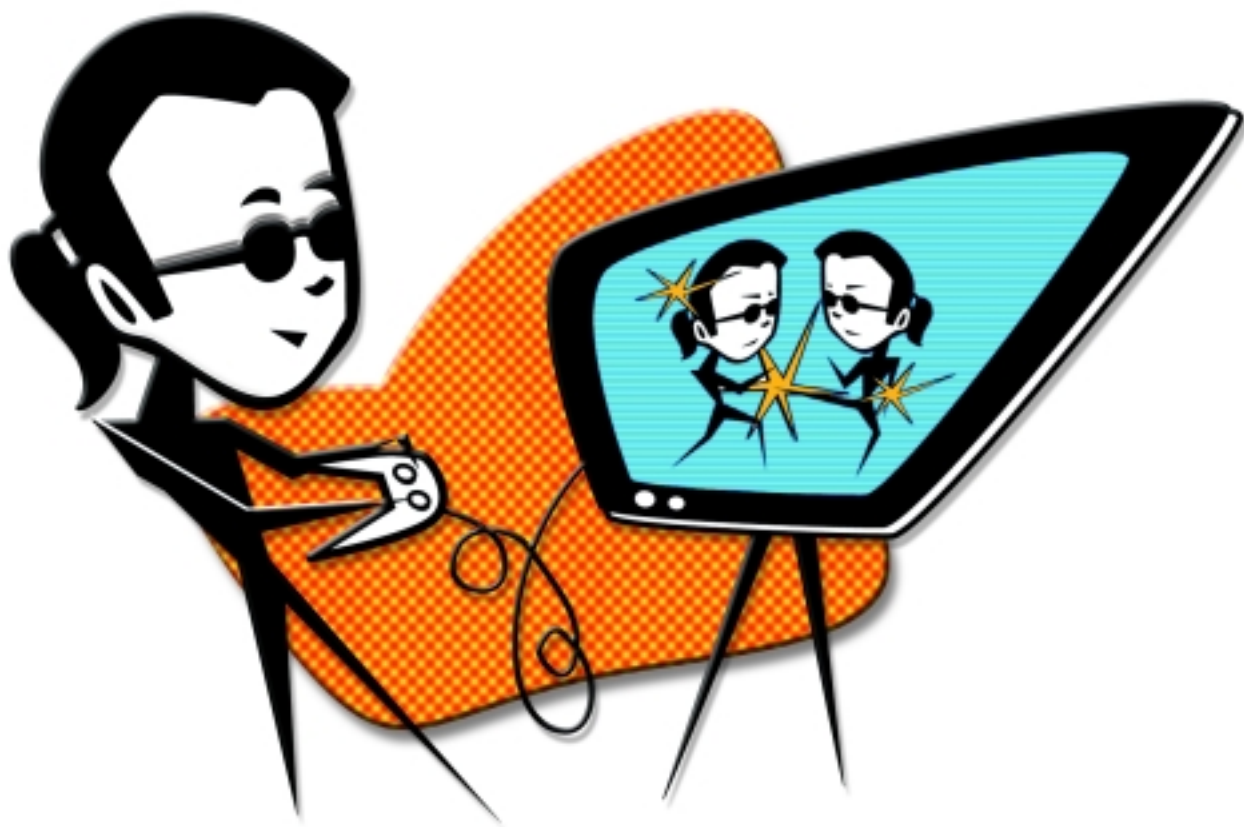
Getting stuck. Unfortunately, even the best CCP can occasionally get stuck, just like the player does. When a CCP gets stuck, it throws its car into reverse, realigns with the road target, and then goes back into drive and resumes the race.

The Road Ahead

In the wake of the original MIDTOWN MADNESS, we wanted open city racing to give players much more than the ability to drive on any street and across any open area. In order for a city to feel and play in the most immersive and fun way possible, many interactive entities of real cities need to be simulated convincingly. These entities include racing opponents, tenacious cops, ambient traffic, and pedestrians, all of which require powerful and adaptive AI to bring them to life. MIDTOWN MADNESS 2 and MIDNIGHT CLUB expand on the capabilities of these entities, which in turn raises the bar of players' expectations even further.

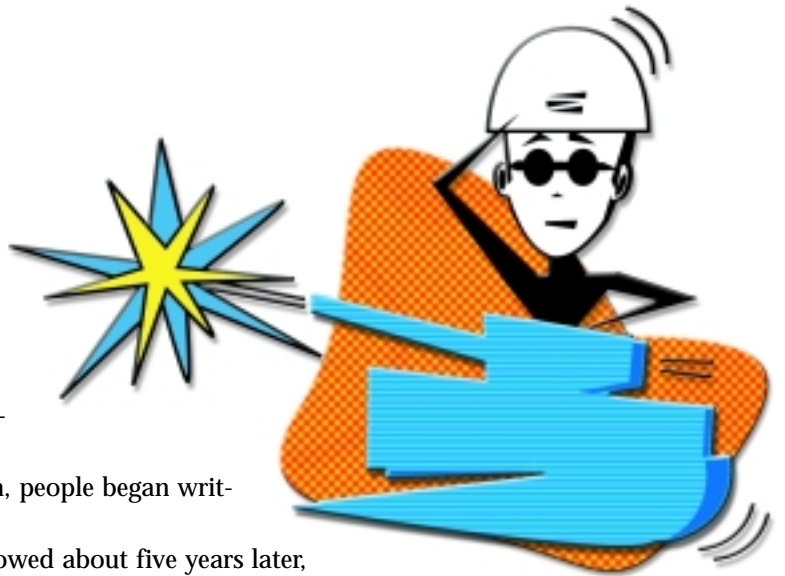
The future of open city racing is wide open — literally. Angel Studios and I are planning even more enhancements to the AI in any future games of this type that we do. Some ideas I'm planning to investigate in the future include enhancing the opponent navigation skills of all AI entities, and creating AI opponents that learn from the players. Additionally, I'd like to create more player interaction with the city pedestrians, and have more interaction between AI entities. Anyone wanna race? 🏁

THE Four Myths OF Game Design



ERNEST ADAMS | *Ernest is an American freelance game designer currently living in England. He was most recently employed as a lead designer at Bullfrog Productions, and for several years before that he was the audio/video producer on the MADDEN NFL FOOTBALL product line. He has developed online, computer, and console games for everything from the IBM 360 mainframe to the Playstation 2. His e-mail address is ewa@earthling.net, and his professional web site is at <http://members.aol.com/ewadams>.*

Thirty-five years ago, when time-sharing operating systems appeared for mainframes and the print-



ing terminal became common, people began writ-

ing computer games. The game industry followed about five years later,

if we count the earliest arcade machines. In the course of the last three decades, we've wandered

down some strange paths, hit a few dead ends, and witnessed the evolution of a new entertainment

medium, perhaps even a new art form.



We've also slowly evolved a discipline of sorts, a

way of thinking about games and the

people who play them. Much of this

accepted wisdom is accurate, tuned by many years of

trial and error and filtered by the natural selection of the

marketplace. For example, we now know that great

graphics alone do not ensure a game's success — the few games which violated that principle did

not survive or reproduce. But a few bad ideas have managed to

hang on as well, perhaps because they're not quite lethal

enough to kill a company that relies on them. In

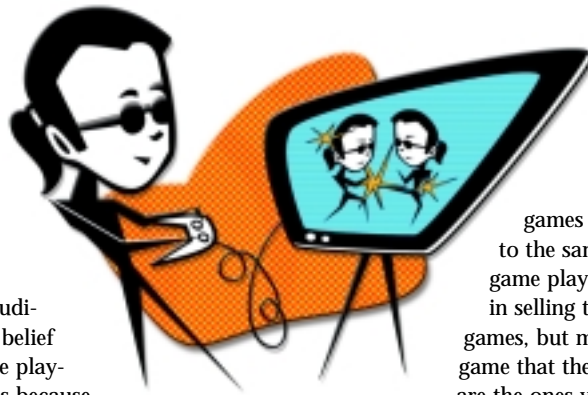
this article, I'm going to discuss four commonly

held beliefs about games and game design that are

erroneous. If we could get these ideas out

of the gene pool, we'd all be better off.





Myth #1: We Are Our Own Audience

The idea that we accurately represent our audience is the foundation of just about every belief that we developers hold about games and game players. We think we know what our audience likes because we know what we like. After all, we're not just game developers, we play games too, and we're convinced that this provides us with the insight to understand all players — and potential players — everywhere.

How many design discussions have you attended where somebody, in criticizing an idea, started a sentence with the words "Nobody cares about . . ."? If you've spent any time in the industry, the answer is probably in the dozens. Nobody cares about history (so the vast majority of games are set in science-fiction or fantasy worlds). Nobody cares about acting (so the acting in most games is abominable). Nobody cares about the story, everyone clicks through it (so the plot is trivial and the text is badly written). Girls don't play games (so very few games of interest to girls are produced). The list goes on and on. The basis for these sweeping statements is seldom any concrete evidence; it's just a belief that as game developers, we know what players want.

This logic is profoundly flawed. We may play games ourselves, but we are a peculiar class of gamers: those who also happen to be game developers. We don't represent those players who *don't* want to make computer games; there aren't any of them among us. We can't assume that our interests are the same as theirs.

Computer gaming is unique among entertainment media for the number of people that it inspires to want to make it their career. Most people watch TV without wanting to produce TV shows, and visit fairgrounds without wanting to run the carousel, yet a surprising number of people who play computer games also want to make them. Why should that be? It's because the games they're playing are designed for the kinds of people who are interested in making games — that is, they're designed by developers, for developers, or at least potential developers.

In my experience, market research in this industry is little more than a joke — and I used to work for one of the most successful publishers in the business. There are a few nods in that direction; every now and then somebody will collect up all the warranty cards returned by the purchasers and read what they've said, but any decent statistician would laugh out loud. Warranty card returns come from a tiny, self-selected minority of the customers. All they tell you is what the sort of person who returns warranty cards thinks — hardly a random sample. Oh, and we hold focus groups, but who do we invite to focus groups? Experienced gamers — specifically, the kinds of players who would enjoy spending an evening bending a publisher's ear. Again, not a terribly representative group. Other than that, the market research I've seen has been based on little more than hunches, conventional wisdom, and Myth #1.

Worse yet, there's another group of people we're ignoring entirely: the ones who don't play any games at all. Right now, we developers are all brutally clawing each other to sell more of the same kinds of

games in the same limited shelf space to the same limited market of current game players. The real opportunity lies in selling to people who don't yet play games, but might start if they could find a game that they liked. These "proto-gamers" are the ones we should be reaching out to,

the people we should be trying to create products for. But we don't know anything about them. All we know is they're not buying the games that we're making now — the kinds of games that we like.

There's a certain number of kids, mostly boys, who hang around the game store and buy a \$40 game every few weeks. They're our traditional market. But there is a staggeringly huge number of people, mostly adults, and many of them women, who would like to take a few minutes between tasks at work or chores at home to play a light, fun, simple game that costs them a few cents, tops. Who's selling to them? Not most of us, that's for sure.

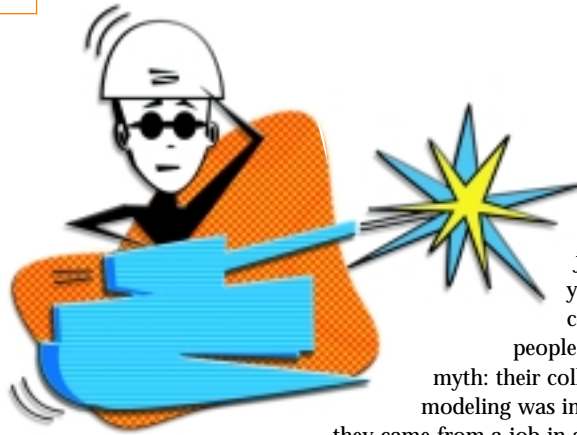
When I mention this notion to developers, especially young ones, I usually get a disgusted look and a flat dismissal: "I won't work on any game that I wouldn't want to play." The more thoughtful sometimes add, "I'm afraid I wouldn't do a very good job if I weren't passionate about the game." I sympathize with that notion — I, too, was one of those people who felt passionate about making games within five minutes of playing one for the first time — but ultimately it's short-sighted. It might be good art, but it's bad business. It still leaves you making games for game developers, and that's an overcrowded field. Passion's difficult to maintain when the game you spent 18 months building lasts three weeks on the store shelves.

Consider *BARBIE FASHION DESIGNER*. *BARBIE FASHION DESIGNER* was not constructed by ten-year-old girls. No offense to ten-year-old girls, but very few of them have the wherewithal to produce commercially viable entertainment software. *BARBIE FASHION DESIGNER* was developed by adults who were fairly unlikely to play with it much themselves. But in spite of that, they did an excellent job and they made a ton of money for themselves and their company. Not knowing the developers personally, I can only assume that they relied not on passion for playing the game itself, but on a different quality called professionalism, the desire to do a job well for its own sake. And it paid off in spades.

The way to overcome Myth #1 is to do something I have seldom seen done in the game industry: decide who your audience is up-front. Don't assume that you're selling to the same jaded old crowd and that you know exactly what they want. Instead, define your audience, then admit your ignorance about them. Go find some of them and actually ask what games they'd like to play, and where and how they want to play them. Who knows, you could discover a huge market that has been ignored for the last thirty years. Jackpot!

Myth #2: Realism Is Always a Primary Design Goal

What is a primary design goal? Here's how to find out. Make a list of everything you want to achieve with your game. These goals can be creative, technical, financial, anything. If you want your game to change the way the player thinks or feels, they



could be intellectual, emotional, or spiritual. They even could be political or social, if you want your game to change the world.

Now sort your list from the most important goals to the least. Then start from the top, run down the list, and ask yourself of each goal, "If we don't achieve this, will we consider the game to be a failure?" At some point, you'll stop saying "yes." Your attitude will change from "this goal is essential," to "this would be nice but isn't critical." At that point, draw a line across the list.

All the goals above the line you just drew are your primary design goals. They're the things you absolutely must achieve. The goals below the line are secondary, things you'd like to include but you don't feel you have to, and certainly not if doing so interferes with anything higher on the list. Secondary goals can be sacrificed for the sake of primary goals.

If you read enough game hype — advertising, box copy, press releases — you could quickly form the impression that realism is one of the most important features of any game. Publishers' marketing and PR departments are constantly harping on the subject, and if you pay attention to such things you might start to believe it yourself. Don't. It's Myth #2.

Realism can and should be a primary design goal in certain genres where it matters. High-end flight and racing simulators demand realism. Few other games need it, and there are some genres in which concentrating on it is actively detrimental. Many games are set in an artificial, symbolic world, and they should be. Monopoly has very little to do with the realities of buying real estate, and if you were to include those details (taxes, insurance, inspections, termi-tes . . .) they would harm the game.

The primary source of this myth is pretty obvious: it's our own history. The audio, video, and processing capabilities of our machines have been continuously improving ever since the first computer game was written, and as a result, games are more realistic now than they ever have been in the past. More importantly, at every point in our history, the games have been more realistic than they ever were before. We're at the top of a steadily rising curve, and we always have been.

Of course these improvements look good, they sound good, and they serve to demonstrate technical competence and advancement. But they will occur automatically if your development team is taking advantage of the target hardware. Remember: primary design goals are those for which you are prepared to sacrifice something else, if necessary. If realism is a primary design goal, what are you willing to sacrifice for it?

Sometimes the answer is playability. Spectrum Holobyte's F-16 FALCON is an extremely realistic flight simulator, and when you play it in that mode, you find out why very few people are good enough to be fighter pilots. It's damned hard to play. Realism is F-16 FALCON's claim to fame, its *raison d'être*, so it's appropriate for realism to be a primary design goal, even at the expense of playability. But it's not appropriate for most other kinds of games.

Industry veterans probably don't need to be told this. The people

who need to hear it are newcomers joining the industry from elsewhere, either young people fresh out of college, or people coming in from other industries. Those people are in fact the secondary source of this myth: their college professors taught them that accurate modeling was important in software engineering, and if they came from a job in another industry, it probably was. The guy who was brought in to head up EA's 3DO development team was a Ph.D. physicist, and he insisted that JOHN MADDEN FOOTBALL for the 3DO must have "realistic" physics. Unfortunately, until we changed his mind, this made the game unplayable. Because the athletes in a sports game are being guided by a simplistic game controller, you have to adjust the physics to compensate, but he didn't understand this. I was the designer of MADDEN at the time, and I taught the programmers a mantra to chant when they got in fights with their boss: "It doesn't have to be 'right,' it has to look good and play well." Eventually we won him over.

There's a legend from the early days at Atari that was told to me by someone who was there at the time. The arcade classic BATTLEZONE had come out, and the U.S. Army sent some people around to find out how Atari was making tank simulators for a few thousand dollars when they were paying millions for theirs. There was a meeting with a lot of brass hats on one side of the table, and a lot of long-haired, T-shirted, dope-smoking programmers on the other. The Army wanted to know how they achieved precision on such low-end gear. The programmers shrugged. "We don't," they said. The officers persisted. "So if an enemy shot should really miss your tank, but the computations are off and it hits it anyway . . ." "The player loses his quarter," the programmers said. "Big deal. He's not going to know, is he?" The Army decided to keep its own simulators.

Like most legends, the exact details may not be right but it's the message that matters, and it nicely illustrates my point about Myth #2. For the Army, realism was a primary design goal — a matter of life and death, in fact — and they sacrificed a lot of money to achieve it. For Atari, fun and manufacturing costs defined the primary goals, with realism a distant second. But BATTLEZONE was a great game for all that. This is an entertainment industry. Don't get needlessly bogged down with realism.

Myth #3: You Can Build a Hit by Imitating Another Hit

This myth tends to be believed more by business people than by designers. Designers usually want to innovate, not imitate. Still, if you look at the store shelves, there's a heck of a lot of imitation going on. It happens because a publisher's marketing and sales people notice that some competitor has had a hit, and they persuade the management that if the developers will just produce something like it, they can have a hit too. They're victims of Myth #3: the belief that you can build a hit by imitating another hit.

I'm sure we've all seen this myth at work. Someone will produce a brilliant new game, it'll be a massive hit at Christmas, and by the next E3 there'll be four or five schlocky knockoffs in the pipeline made by other people. They never look as good, because



chances are they're being rushed to market to catch the coattails of the original, and they never sell as well, because the "wow" factor is already gone. Why do they bother? Well, you can make a little money that way if you've got no pride and no creativity of your own. But in my experience the companies that do this are also-rans, second-rate outfits that will never really shine. For one thing, if the management has foisted it on the developers against their will, they are wasting their own talents. They have got their people building cheap knockoffs when they could be working on something innovative and new. No developer with any imagination is going to put up with that for long. They'll leave, and then the company has to find someone to replace them who doesn't mind working on cheap knockoffs. It's not a formula for building and maintaining an excellent staff.

Oddly enough, this can even happen within a single company. It occurs when the marketing department insists on creating a sequel to a game for which no sequel was intended. Sometimes a game is a hit because the development team has burned themselves out, put everything they had into that one game. If you then demand that they make another just like the first, you're bound to get an inferior product — they don't have anything left to give. With the current emphasis on franchises, we're usually better at product planning than that. But it still happens, especially with games that were unexpectedly successful.

I don't mean to suggest that you shouldn't study other people's games. I'm a firm believer in the value of studying other people's games; heck, I'm a firm believer in the value of studying everything. It's all grist for the creative mill. But there's a significant difference between keeping an eye on the competition and ripping them off. The latter is seldom successful. Our objective should be to surpass the competition, to create "wow" moments of our own, rather than hoping for a free ride on the back of someone else's imagination.

Myth #4: A Great Idea Will Make You a Fortune

Several times a year, I get letters from people who have great game ideas, but no clue how to make them a reality. They'd like me to teach them all about game development and marketing, but they're usually very cagey about what their idea is — they're afraid I might steal it and make a lot of money that's rightfully theirs. Alas, they've been seduced by Myth #4: the notion that a great idea will make them a fortune. It's a classic among fledgling game designers, so this section is for them.

Part of the reason people believe Myth #4 has to do with the way we're taught about the history of innovation. We're told neat little sound-bite chunks of history that don't include the whole story, things like "James Watt invented the steam engine." This gives the impression that in a world of horse-drawn coaches, James Watt saw the lid of his teakettle jiggling and suddenly the railroad was born. But James Watt was part of a much larger process, and what he really invented was a technical improvement

to existing, stationary steam engines. He didn't invent the railroad, either. It was a man named George Stephenson who constructed the first steam locomotive — drawing on the work that Watt had already done, of course.

It's not that a great idea won't ever make someone a fortune. Sometimes one does, and a lot of people point to TETRIS as the perfect example. The problem is that truly great ideas on the order of TETRIS are extremely rare. For every TETRIS there are tens of thousands of seemingly great ideas that go nowhere. A winning lottery ticket will make you a fortune, too, but if you're serious about making money, lottery tickets are not the way to do it.

Another game that people point to as an example of a great idea that made a fortune is DOOM. Everyone remembers that DOOM was like nothing ever seen before, and it enabled John Carmack and John Romero to buy Ferraris. But DOOM wasn't actually a great idea from out of the blue; in fact, it's an excellent example of how fortunes are really made in the game industry.

The central idea of DOOM — running around and shooting things in the first person — was not new. There was a multiplayer game called MAZEWAR on the Xerox Alto word processor as far back as 1983, and there are probably earlier examples. The central display method in DOOM, a technique called raycasting, was not



new either. The reason DOOM did so well was not because it was great new idea, but because it was a brilliant execution of a variety of existing ideas. Raycasting may not have been new, but DOOM did it better than anyone else. The game was so simple, so fast and clean, that it still has its devotees today. I've seen it ported to all kinds of strange devices, even the LCD display of a digital camera.

That's the thing to know about fortunes in the game industry. There's nothing wrong with great ideas, but it's a mistake to think that they routinely lead to fortunes. What reliably makes money is not brilliant ideas, but quality workmanship: top-notch, first-rate, class-A execution. And the only way to obtain that is by the sweat of your brow.

Your chances of selling a great idea to a publisher just in idea form are rapidly vanishing. Once upon a time publishers might have bought ideas, but they don't anymore — and in any case, publishing companies are full of people who all have their own great ideas. Unless yours is of winning-lottery-ticket caliber, nobody's going to pay you for it. What publishers want is not ideas, but people who can create products, especially with the kind of quality that I was talking about.

But suppose you're still convinced that yours is a lottery-jackpot idea. How do you protect it? Here's a one-paragraph primer on intellectual property for wannabes. In America there are three ways to legally protect your work: copyright, trademark, and patent. A copyright protects a document of some kind, either text, a photograph, a sound recording, or some other expressive material. It doesn't protect the ideas in the document, only the document itself. You can't copyright an idea. A trademark is a name, slogan, or logo that you use to represent your company or its products. Trademarking something prevents other people in the same line of business from using it as well. You can't trademark an idea, though; again, it has to be something concrete. Finally, a patent is a means of protecting a new method for doing something. No one else must ever have done it before, and it actually has to be a method, not something abstract like a story or a character. You can patent an idea, but it has to be an idea about doing something, not just an image or a concept.

So if you have a game idea like "Robot Camels from Neptune Invade the Justice Department," you can't copyright, trademark, or patent it. You can draw a robot camel and copyright your drawing, and you can name the camels "Dromedroids" and trademark the name, but other than that you can't prevent other people from having and developing the same idea. Anybody else can make a game on the same subject. If you come up with some method of playing your game that has never been seen in any other game ever invented, you might be able to patent the method, but you still can't patent the robot camels. Besides, obtaining a patent is an expensive and time-consuming process. The burden of proof is on you to show that your method is so different from anything that anyone else is doing that you deserve exclusive rights to the idea. With game mechanics, that's going to be a tough sell.

The one other option is to treat the idea as a trade secret — that is, not to tell anybody about it, and if you do tell someone about it, to ask them to sign what's called a nondisclosure agreement, or NDA, first. An NDA is usually

a simple, one-page contract in which somebody promises not to reveal your secret to anyone else, in exchange for getting a chance to look at it. Normally, no money changes hands. This is what I use when people want to consult me but don't want to tell me about their idea. I sign a nondisclosure agreement promising not to reveal their secret or use it for myself. However, getting signed NDAs doesn't take you any farther down the road to that hypothetical pot of gold, either.

All this may make it sound as if I think there's no point in innovation, and that we might as well go on making the same kinds of games because great ideas are worthless. Nothing could be farther from the truth. Great ideas are wonderful, but you need a realistic understanding of their value. No single idea is likely to be worth a fortune, so rather than clinging to it as if it were a diamond, we should continually generate new ones: learn, think, dream, create.

A couple of weeks ago I had the opportunity to visit a company called Hidden Dinosaur in Sweden. Its founder, Michael Stenmark, showed me his sketch book of ideas for the project they're working on. To say that I was amazed would be an understatement — overwhelmed is more like it. He had page after page of places, people, creatures, objects, stuff I had never seen the likes of. There wasn't an elf, samurai, or space marine in sight. Everything was fascinating and new, and every day he goes out and draws more things. He often travels for inspiration, and he never stops. He doesn't assume one idea is enough. They pour from his pen like a rainbow Niagara. He's got the right attitude about ideas: more is better.

If you've got a great idea, use it to practice your skills. Learn how to develop it. If you want to be a programmer, learn to program it; if you want to be an artist, learn to draw it; if you want to be a writer, write about it — and you can copyright all the material you create, so at least your labor is protected. You won't make a fortune, but if you work on it, your passion for it will show. Then you can bring it out at job interviews to demonstrate your talent. Don't worry about keeping it secret. In fact, do the exact opposite: talk about it, to anyone who will listen. If it's really that good, they'll be impressed with your imagination and more inclined to hire you.

In Conclusion

As I said, these four myths aren't lethal mutations. Believing them isn't necessarily going to destroy your career or your company. But like flaws in the genetic code, they accomplish nothing, they do more harm than good, and it's undesirable to pass them on to the next generation. By identifying and correcting them, perhaps we can effect a few repairs on our rapidly growing industry. 🐾

Ritual Entertainment's HEAVY METAL: F.A.K.K. 2



PUBLISHER: Gathering of Developers

PROJECT LENGTH: 18 months

NUMBER OF FULL-TIME DEVELOPERS: 11-18

NUMBER OF CONTRACTORS: 1

BUDGET: Approx. \$2 million

RELEASE DATE: July 31, 2000

PLATFORMS: Windows 95/98/NT

DEVELOPMENT HARDWARE USED: 450MHZ

Pentium IIs with 256MB RAM, TNT2

Ultras, and 13GB hard drives.

DEVELOPMENT SOFTWARE USED: Visual C++

6.0, Photoshop 5.0, 3D Studio Max 3.1,

Deep Paint 3D

NOTABLE TECHNOLOGIES: Quake 3 engine from

id Software, RAD Game Tools' Miles

Sound System, InstallShield 6.2,

DemoShield 6.51, Robocopy 1.96 (from the

Windows NT Resource Kit)

PROJECT SIZE: 364,825 lines of code;

11,519 files

The decision to create a game based in the *Heavy Metal* universe started back in the hot Texas summer of 1997. Ritual Entertainment was in full-blown production of *SIN* when our concept artist's agent, Russell Binder, called up one day. Russell mentioned that his client, Kevin Eastman, was looking to make a videogame based on a new animated movie called *Heavy Metal: F.A.K.K. 2*. Kevin told us that the name of the movie was based on the name of the lead character in the movie and that character was being based on the image of his wife, Julie Strain. The name of the movie was later changed to *Heavy Metal 2000*, but we decided to keep the *F.A.K.K. 2* name, due to the fact that we already had two magazine covers and previews, and we didn't want to create product confusion.

Ritual jumped at the chance to create a game based in the *Heavy Metal* universe. Everyone at Ritual was familiar with the original *Heavy Metal* movie released in 1981, and a lot of us grew up reading the *Heavy Metal* illustrated magazine. A few business meetings later with Kevin and Russell brought the deal to a close.

Ritual then hired a new development team consisting of two programmers and two level designers. This small team worked on preproduction of *F.A.K.K. 2* during production of *SIN* in 1997 and on into the beginning of 1998.

Circumstances arose that caused the two programmers to leave Ritual, and the level designers were assigned to the *SIN* team. This caused *F.A.K.K. 2* production to slow down during *SIN*'s final crunch, from March to October 1998. After *SIN* shipped, part of the team worked on *SINCTF* (a free multiplayer add-on to *SIN*) while the rest of the team moved over to work on *F.A.K.K. 2*. When *SINCTF* was released to the Internet, the *F.A.K.K. 2* team was fully realized. We started off with five programmers, five artists, five level designers, one support person, one sound engineer, and one project manager. However, when we finished the game the core team consisted of three programmers, three artists, three level designers, one sound engineer, and one support person. Art director Robert Atkins was the only team member that survived from the beginning of preproduction to the very end of the project when *F.A.K.K. 2* went gold on July 31, 2000.

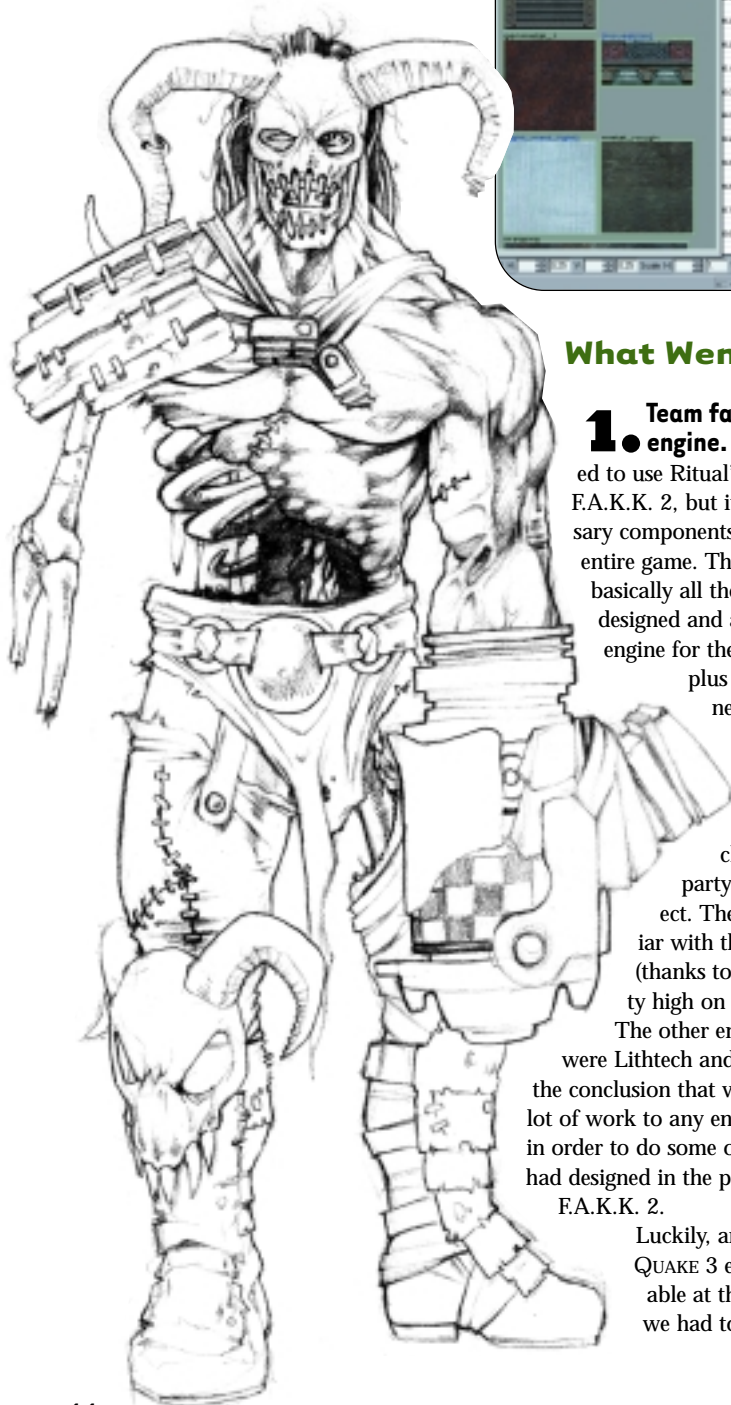
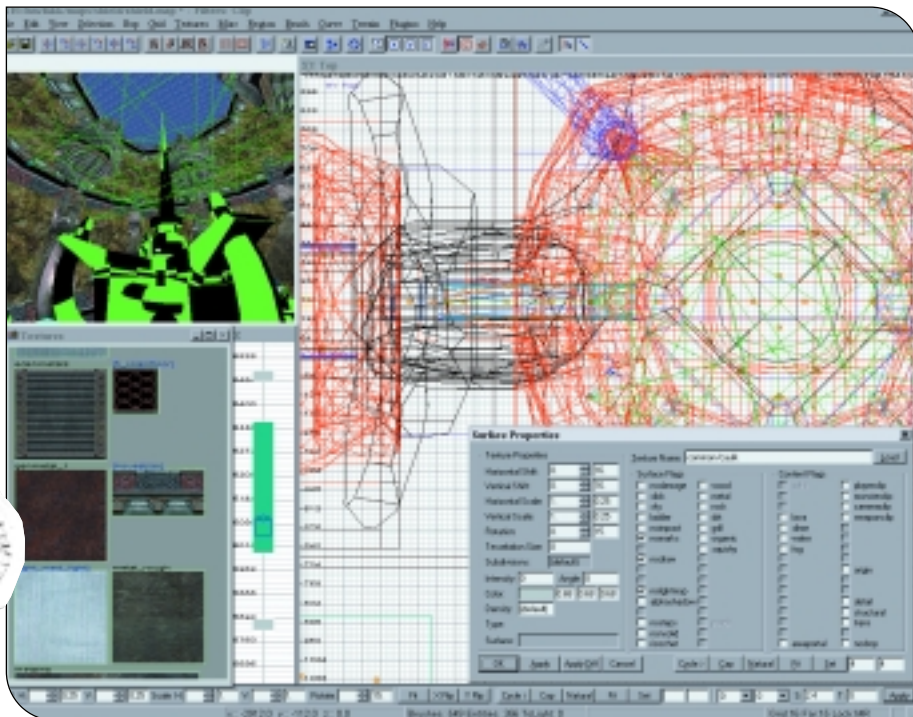
In the midst of this turnover, many of the original design team left, and many elements of the original design laid out in preproduction had to be rewritten. *F.A.K.K. 2* production began in earnest in early March 1999 with the new design.

SCOTT ALDEN | Scott is currently working as a programmer on *DUKE NUKEM FOREVER* at 3D Realms. He was one of the senior programmers on *F.A.K.K. 2* and *SIN* for Ritual Entertainment. Scott started his career in the gaming industry at 3Dfx Interactive, where he provided support for the Voodoo 1 graphics card. He can be contacted at scotta@3drealms.com.



THIS PAGE. Julie's nemesis,
Lord Tyler, on a good day.

RIGHT. Q3Radiant level design tool showing the Shield Generator puzzle. BELOW. Concept sketch of Lord Tyler (final boss).



What Went Right

1 Team familiarity with the engine. We originally intended to use Ritual's UberEngine for F.A.K.K. 2, but it lacked some necessary components to complete the entire game. The UberEngine was basically all the components we designed and added to the QUAKE 2 engine for the development of SIN, plus a new renderer and networking layer. There was still a lot of work to be done on the UberEngine, though, and we chose to license a third-party engine for the project. The team was very familiar with the QUAKE 2 engine (thanks to SIN), and it was pretty high on our list of choices. The other engines we considered were Lithtech and UNREAL. We came to the conclusion that we would have to do a lot of work to any engine (besides QUAKE) in order to do some of the things that we had designed in the preproduction of F.A.K.K. 2.

Luckily, an early version of the QUAKE 3 engine became available at the last moment before we had to make a final deci-

sion. We ultimately ended up with a limited QUAKE 3 license where we got a snapshot of the code in February 1999. QUAKE 3 had not been released yet, but it was in a workable state when we first got the code. This was the best decision we made during the entire project.

Ritual Entertainment is extremely familiar with the QUAKE family of engines, having started out with the QUAKE 1 engine on the QUAKE MISSION PACK #1: SCOURGE OF ARMAGON. Next, SIN was initially developed under a modified QUAKE 1 engine and was converted over to QUAKE 2 late in the project. The team was very relieved that we wouldn't have to add the extra learning time that it would take for us to become familiar with Lithtech or UNREAL. The level designers would not have to learn a new editor, and since they were already familiar with the capabilities of the QUAKE engine, they could build prototype levels very quickly at the beginning of the project.

From the programmers' standpoint, it let us leverage all of the code that we wrote during our development of SIN. We were able to drop in the SIN game code and get a working version of the game very quickly, including our scripting language.

We received the QUAKE 3 code in late February 1999 and had a very impressive technology demo ready for E3 in early May. Since we had five programmers at the beginning of the project, in the early

months of the project we were able to drop tons of new features into the engine and show them off at E3.

2. In-game tools and engine modifications. The modifications that we made to the QUAKE 3 engine helped us put on the finishing touches and ultimately garnered F.A.K.K. 2 such praises as “The Most Beautiful Game Ever.” After hearing some of those comments, all the extra work required really was worth it. Almost all of the tools that we created on top of the QUAKE 3 engine allowed the level designers and artists to create content with a text file. When a system was designed, we always took into account how the team was going to use it. We used text files and simple English commands to allow the designer or artist to make game changes without having to recompile anything.

Another element that we added to the tool system was the ability to edit things in the game engine itself. This saved a lot of time for the artists and level designers. They got to see their changes immediately on the screen instead of having to exit out of the game and restart every time they changed something.

Skeletal animation system and LOD. One of the first modifications to the engine was the addition of a skeletal animation and level-of-detail (LOD) system. This allowed us to put in a ton of animations for the game’s main character, Julie. We could also throw a lot of in-game models and monsters on the screen thanks to the LOD system. Our lead animator, Darrin Hart, hand-animated 11,000 frames of animation, of which about 6,500 frames were actually used in

the game. This would not have been possible with a vertex-based animation system.

Morpheus scripting language. One of the best systems we came up with during the development of SIN was the scripting system. We gave it the internal nickname Morpheus, partly in tribute to *The Matrix*, and partly due to the fact that you could do just about anything you wanted to with it.

The scripting system is described in more detail in the Postmortem of SIN I wrote (March 1999), but in a nutshell the scripting language gives the level designer complete control over any object or entity in the game. The functionality it provides ranges from the simple linear movement of objects around the level to the complex scripted sequences that exist throughout F.A.K.K. 2. We

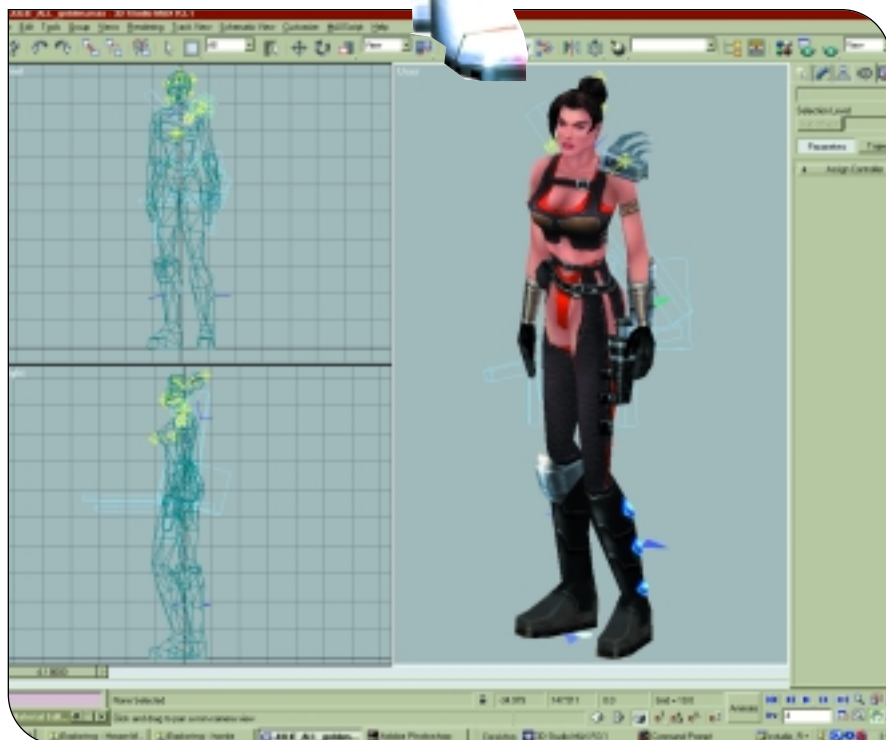
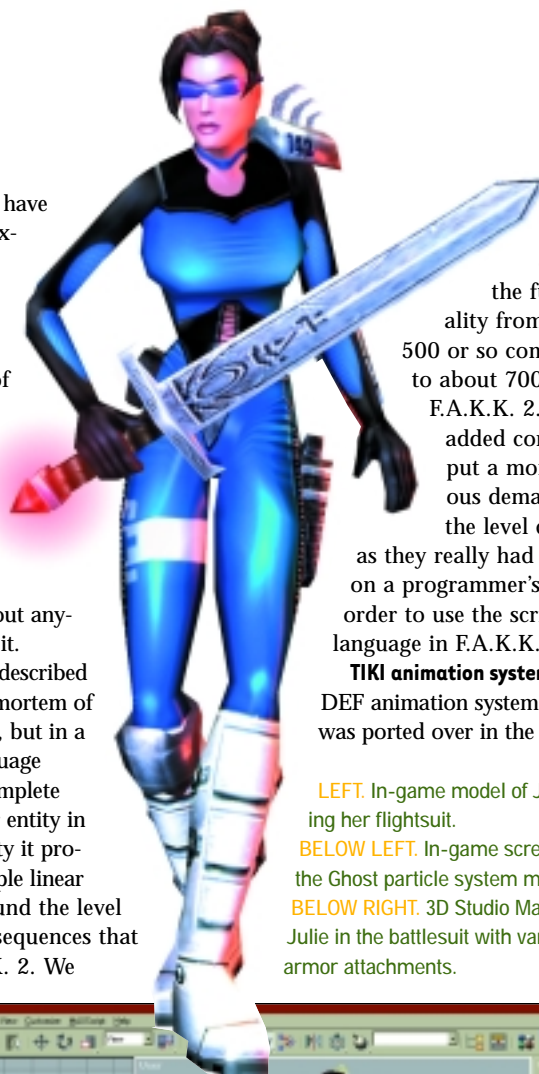
extended the functionality from SIN’s 500 or so commands to about 700 in F.A.K.K. 2. This added complexity put a more rigorous demand on the level designer, as they really had to put on a programmer’s cap in order to use the scripting language in F.A.K.K. 2.

TIKI animation system. The DEF animation system from SIN was ported over in the early

LEFT. In-game model of Julie wearing her flightsuit.

BELOW LEFT. In-game screenshot of the Ghost particle system menu.

BELOW RIGHT. 3D Studio Max showing Julie in the battlesuit with various armor attachments.





stages of F.A.K.K. 2 development and renamed TIKI. Its main function was to allow the text file definition of models in the game. It also allows events to be synched with the animation on a per-frame basis. It was largely used for synching sounds and effects with the character animation.

Ghost particle system. The Ghost particle system was written completely from scratch and incorporated into the Morpheus scripting language and the TIKI animation system. Ghost allows artists and level designers to create user-defined particle systems and integrate them into the game via the animation system. As is the case with almost every Ritual-created engine modification, this is accomplished with a simple text file. There are about 50 commands used to modify the parameters of a particle system, and the systems can be combined together to create some complex-looking effects. For example, the firing of the Uzi in the game is a combination of five particle systems: smoke, shells, muzzle flash, tracer, and impact debris.

Cinematic camera system. F.A.K.K. 2 is a very cinematic-intensive game. The story demanded some very complex scenes to advance the story, and we wanted to show as much detail as possible in these cinematic elements. We started with the spline-based camera system in *SIN* and ported it over to F.A.K.K. 2. We also added the ability to edit the camera paths in the game, actor tracking, and field-of-view control. This gave the cinematic designer complete control over the scene and allowed him to preview his changes immediately.

Sound system (Zound). Another in-game system is the Zound editor. This system let our sound engineer place sound and music triggers in the game without having to recompile the entire level. This allowed him to place music cues and music mood triggers around the level to create tension or humor when needed.

3 • Third-party license and creative control. Kevin Eastman is the owner of the *Heavy Metal* magazine and co-creator of the Teenage Mutant Ninja Turtles. Together with Simon Bisley, they were the creative force behind the *Heavy Metal 2000* movie. When we discussed the possibilities of the game, we came up with idea of the game being a sequel to the movie instead of doing the standard movie-to-game conversion. He was thrilled to hear this, and fully supported our decision to make the game a sequel to the movie.



TOP. In a scene from *Heavy Metal 2000* Julie convinces Germaine St. Germaine to help her out.

BOTTOM. The Ghost particle system in action.

Very early in the production we received a crate full of *Heavy Metal* magazines — every single issue. We spent days poring over the issues to see what kinds of styles *Heavy Metal* is known for, and to get inspiration for designing the characters.

Kevin also gave us complete control over the story and provided us with some inspirational concept sketches, which influenced the design of the game. Working with Kevin has been a blast and he's supported us completely with our decisions on what to put in the game. For instance, when we decided to resurrect Lord Tyler from the movie, Kevin wholeheartedly agreed.

4 • Focus on single-player. Early in the development of F.A.K.K. 2, we decided to focus solely on the single-player aspect of the game. This was a very tough decision for the team, as just about everyone at Ritual loves to play multiplayer games, and *QUAKE 3* deathmatch is a frequent pastime around the office.

Since the team was small, we decided to focus on the single-player aspects of the game instead of trying to do multiplayer, which usually has an impact on what can be accomplished in the single-player version. We came up with a very tight game design and avoided repetitive gameplay. Almost every level in the game presents the player with new monsters and weapons.



All of the levels were sketched on paper before the level designers started working. They used these 2D maps to get a feel for how the level was going to be laid out and where all the action and encounters were going to take place. Another part of pre-production that helped out was the detailed description of the game's characters and monsters. The character sheets gave the artists the knowledge of how the creature was going to function in the game, which helped them create the necessary animations.

5 • Strong focus on graphics. Right from the beginning of the project, we decided to focus strongly on graphics. We wanted the visual experience in the game to be something unparalleled in the game industry. Since the game is in third-person perspective, the main character had to look great — if you spent the entire game looking at this character, it should be really nice to look at. We combined elements from the original *Heavy Metal* movie, the comics, and the new movie to create Julie. We went through three revisions of her character before finally settling on what you see in the game.

When we designed the world, we wanted it to be an interactive version of the *Heavy Metal* universe. We decided to use a rich color set with lots of red and yellows in the town, and we used vibrant greens and blues for some of the outdoor areas. We wanted to get away from the dingy, dark look that so many other shooters in the genre have. We put the *QUAKE 3* shader system to great use and were very liberal in our use of weapon effects and cinematics. For our efforts, critics have proclaimed *F.A.K.K. 2* to be one of the best-looking games of 2000.

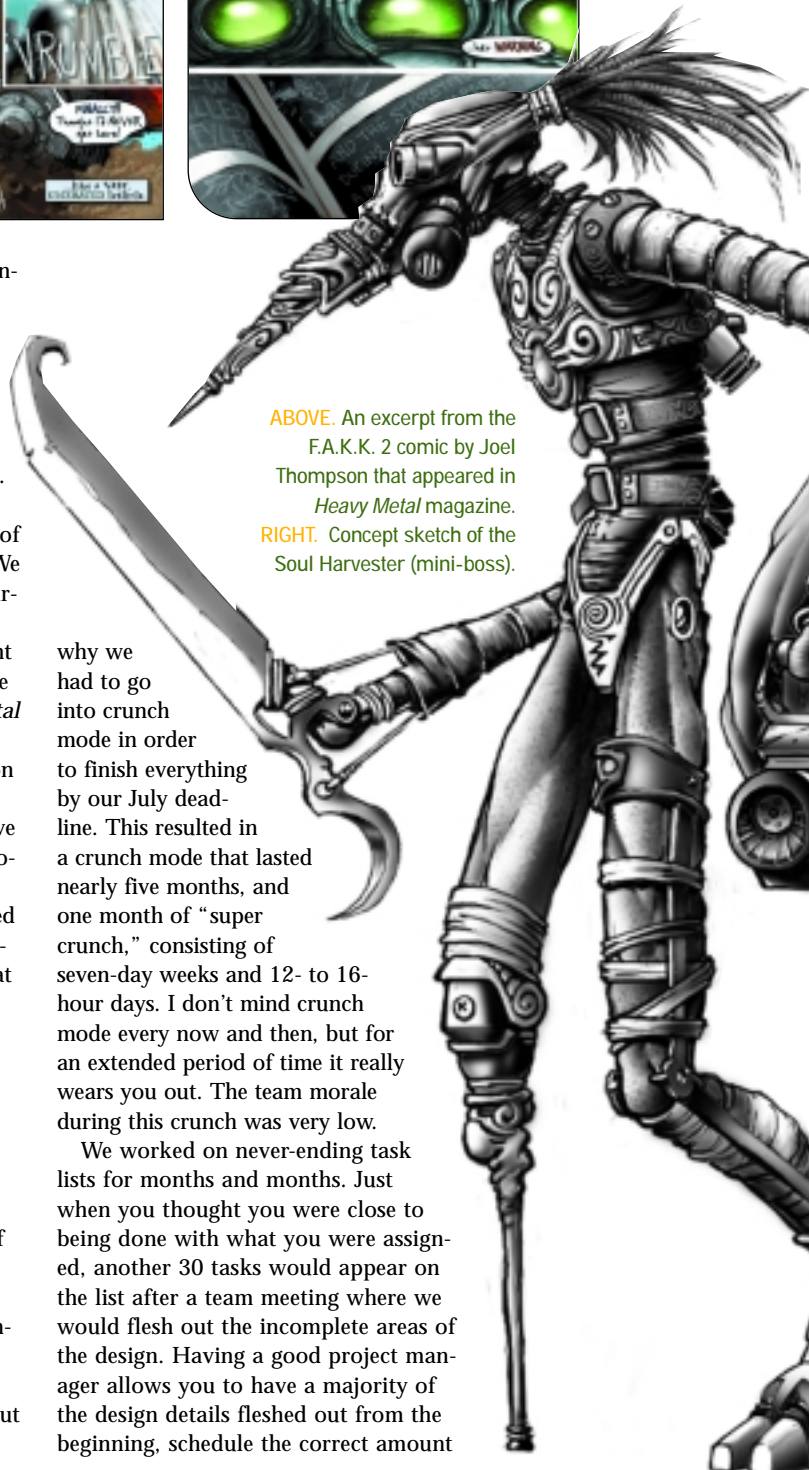
What Went Wrong

1 • No project management. During the development of *F.A.K.K. 2*, we had a project manager at the beginning of the project and a different project manager near the end. There were 12 months of development time in between, during which we had little to no management on the project. There were members of the team that took on this role, but only in a limited capacity, as they had tons of other work to do as well.

Not having a single person manage tasks, and letting just about every feature request get added to the list, was a major reason

why we had to go into crunch mode in order to finish everything by our July deadline. This resulted in a crunch mode that lasted nearly five months, and one month of "super crunch," consisting of seven-day weeks and 12- to 16-hour days. I don't mind crunch mode every now and then, but for an extended period of time it really wears you out. The team morale during this crunch was very low.

We worked on never-ending task lists for months and months. Just when you thought you were close to being done with what you were assigned, another 30 tasks would appear on the list after a team meeting where we would flesh out the incomplete areas of the design. Having a good project manager allows you to have a majority of the design details fleshed out from the beginning, schedule the correct amount



ABOVE. An excerpt from the *F.A.K.K. 2* comic by Joel Thompson that appeared in *Heavy Metal* magazine. **RIGHT.** Concept sketch of the Soul Harvester (mini-boss).



ABOVE. In-game screenshot of Julie's house.

LEFT. In-game screenshot of Julie battling the Happy Mask Hourde creatures.

BELOW. In-game model of Gith Recruiter.



handle on the design of the game. The key designers of gameplay had moved on, and weren't available to talk about the ideas that they had come up with. We ended up scrapping a

lot of the original design document and starting over. This set us back pretty far in the gameplay area. Another area that suffered was models and animations. We lost several artists who worked on different animation packages, and when they left we had to redo the models they were responsible for.

of time for tasks, and have the appropriate number of people on the team to finish a game. This seems to be a major problem in the gaming industry, as nearly everyone I talk to has just about the same story about project management and death-march crunch modes.

2. High team turnover rate on a small team. At the beginning of this article, I mentioned that the team started off with 18 members, and we finished the game with 11. Of those 11 people, only one person was on the original F.A.K.K. 2 team from the beginning. It was a weird project, because most of the team didn't have a

The gaming industry is a turbulent one; people join and leave companies on a regular basis, and it definitely has an impact no matter what anyone says. Unless a person didn't contribute anything to the game, losing personnel greatly impacts finishing a game on time. F.A.K.K. 2 had a very small team that had to work extra hard to make up for the lost employees that weren't replaced. In the end though, we were very satisfied with the game's quality in spite of having lost so many people during the course of development.

3. No multiplayer; a short game for hardcore players. As I said, F.A.K.K. 2 was designed as a single-player game from the start. We were going to try to put multiplayer into the final release if we had the time, but our July deadline came so quickly that we just didn't have the time to finish it. We also designed a very tight game that can be finished by hardcore game players in less than ten hours.

This was our biggest complaint from reviewers and players alike. I do agree that the game is on the short side, but we didn't want to put in dozens of levels that repeated the same gameplay over and over, as so many other games do. Even though I am defending our decision in this article, I do acknowledge it as one of the problems that we had with the design. A short game with no multiplayer has a very limited lifetime in the gaming industry. (Note: As of the writing of this article, a multiplayer patch is in the works that will provide arena-style battles in various F.A.K.K. 2 settings.)

4. No demo before release. Our July deadline was fast approaching, and we had yet to release a playable demo to the Internet. This hurt us in two ways. First, we weren't able to build up any pre-game buzz by having a killer demo for our game, and when the game was released people seemed surprised to hear about it. Second, a lot of people will not buy a game unless they play a demo beforehand to see if they like it.

Fortunately, we were able to get a demo out the door within two weeks of our release, but the jury is still out on whether or not this is a good or bad thing.



ABOVE: Julie solving the puzzle of the Tiki Heads in the Swamp.

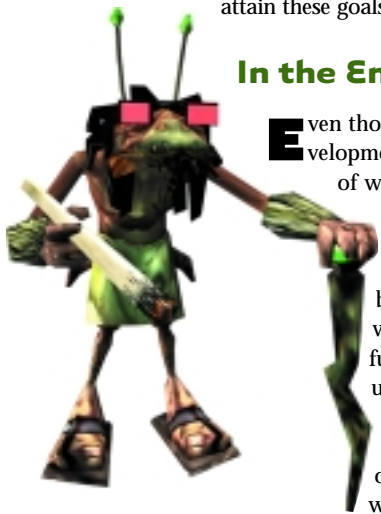
BELOW: In-game model of the wise old Gruff.

5. Complex systems. I mentioned this problem in the SIN Postmortem back in March 1999, and it looks like it hit us again. It's not surprising, since we used many of the same systems that we had in SIN, and we even extended them. Game design on the PC is becoming more and more complex as time goes on. Level designers aren't just creating levels anymore. They have to design puzzles, scripted events, cinematics, and on and on. In F.A.K.K. 2 this is all done through the scripting language, so each level designer had to be something of a programmer as well. On the artists' side, they also got a taste of programming. The effects system is driven entirely by text files, and the artists needed to learn the system's intricacies in order to get the effects they wanted.

This complexity added more to the development time than we had anticipated. Even though everyone on the team was really familiar with the engine, getting used to the new tools and modifications took a lot of extra work. The scripts for F.A.K.K. 2 total nearly 10,000 lines. I'm not really sure if this problem is ever going to change in the industry. As people demand ever more immersive game environments, the complexity goes up in order to attain these goals.

In the End

Even though F.A.K.K. 2 had its share of development problems, we are very proud of what we accomplished with the game. While the design of the game was not always as well defined as some of the team members would have liked, we still were able to create an incredibly fun third-person action game in just under 18 months. We feel that F.A.K.K. 2 is our best game to date, and our future games will only benefit from the experiences we had during its development.



United States Postal Service
Statement of Ownership, Management, and Circulation

1. Publication Title: **Game Developer** 2. Publication Number: **1073-0228** 3. Filing Date: **November 1, 2000**

4. Issue Frequency: **Monthly** 5. Number of Issues Published Annually: **12** 6. Annual Subscription Price: **\$19.95**

7. Complete Mailing Address of Known Office of Publication (Not printer) (Street, city, county, state, and ZIP+4):
**OGP Media Inc.,
 600 Harrison Street,
 San Francisco, CA 94107**

8. Complete Mailing Address of Headquarters or General Business Office of Publisher (Not printer):
**OGP Media Inc.,
 600 Commodore Drive,
 San Francisco, CA 94107**

9. Full Names and Complete Mailing Addresses of Publisher, Editor and Managing Editor (do not leave blank):
Jennifer Pakka, Publisher: OGP Media Inc., 600 Harrison Street, San Francisco, CA 94107
Mark McLoura, Editor: OGP Media Inc., 600 Harrison Street, San Francisco, CA 94107
Sara Haber, Managing Editor: OGP Media Inc., 600 Harrison Street, San Francisco, CA 94107

10. Owner (Do not leave blank. If the publisher is owned by a corporation give its name and address or the names and addresses of all corporations owning or holding a majority of the total amount of stock. If not owned by a corporation, give the names and addresses of all individuals owning or holding 1 percent or more of the total amount of stock. If the publication is published by a partnership or other unincorporated firm give its name and address as well as that of each individual owner. If the publication is published by a proprietor give its name and address.)
OGP Media Inc., 600 Harrison Street, San Francisco, CA 94107
a wholly-owned subsidiary of
United News and Media Inc., Ludgate House, 245 Montgomery Street, London, SE1 9UR, United Kingdom

11. Known Bondholders, Mortgagees, and Other Security Holders Owning or Holding 1 Percent or More of Total Amount of Bonds, Mortgages, or Other Securities. If none, check box: None

12. Name: None Complete Mailing Address: None

13. Willing (For completion by nonprofit organizations authorized to mail at nonprofit rates) (Check one)
 Has Not Changed During Preceding 12 Months
 Has Changed During Preceding 12 Months (Publisher must submit explanation of change with this statement)

13. Publication Title: Game Developer		14. Issue Date for Circulation Data Below: November, 2000	
15. Extent and Nature of Circulation		Average No. Copies Each Issue During Preceding 12 Months	No. Copies of Single Issue Published Nearest to Filing Date
a. Total Number of Copies (Net press run)			
		42,816	43,177
b. Paid and/or Requested Circulation (Sum of 1b1, 1b2, and 1b3)			
1b1 Paid (Through Carriers, Retail, and other means)		24,471	26,275
1b2 Paid (Outside-County Subscriptions, Single Copies, and other means)		0	0
1b3 Sales Through Dealers and Carriers, Street Vendors, Counter Sales, and Other Non-USPS Paid Distribution		1,837	1,741
1b4 Other Classes Mailed Through the USPS		0	0
c. Total Paid and/or Requested Circulation (Sum of 1b1, 1b2, 1b3, and 1b4)		26,308	28,016
d. Free Distribution (Sum of 1d1, 1d2, and 1d3)			
1d1 Outside-County as Listed on Form 3841		0	0
1d2 In-County as Listed on Form 3841		0	0
1d3 Other Classes Mailed Through the USPS		0	0
e. Free Distribution Outside the Mail (Carriers or other means)		0	500
f. Total Free Distribution (Sum of 1d1, 1d2, 1d3, and 1e)		1,394	500
g. Total Distribution (Sum of 1c and 1f)		27,702	28,516
h. Copies not Distributed		5,415	4,661
i. Total (Sum of 1g and 1h)		43,217	43,177
j. Payment of Post and/or Requested Circulation (USPS only) (Form 3841)		96.32	98.72
16. Publication of Statement of Ownership <input checked="" type="checkbox"/> Publication required (Will be printed in the December 2000 issue of this publication. <input type="checkbox"/> Publication not required. 17. Signature and Title of Editor, Publisher, Business Manager, or Owner: <i>Mark McLoura</i> Date: <i>11/1/2000</i>			

Instructions to Publishers

- Complete and file this copy of this form with your postmaster annually on or before October 1. Keep a copy of the completed form for your records.
- In items where the publisher or security holder is a trustee, include in items 10 and 11 the name of the person or corporation if known by the trustee's name. Also include the names and addresses of individuals who are stockholders owning or hold 1 percent or more of the total amount of bonds, mortgages, or other securities of the publishing operation. In item 11, if none, check the box. Use State sheets if more space is required.
- Be sure to furnish all circulation information called for in item 15. Free circulation must be shown in item 15c, 1, and 1c.
- Items 10c, 10d, 10e, and 10f must include (1) newspaper copies originally mailed on Form 3841, and returned to the publisher (2) estimated copies from news agents, and (3) copies for office use, libraries, schools, and all other copies not distributed.
- If the publication had Periodicals authorization as a general or requester publication, this Statement of Ownership, Management and Circulation must be published. It must be printed in any issue in October or, if the publication is not published after October, the first issue published after October.
- In item 16, indicate the date of the issue in which this Statement of Ownership will be published.
- Item 17 must be signed.
 Failure to file or publish a statement of ownership may lead to suspension of Periodicals authorization.

PS Form 3826, October 1999 (Rev. 9/99)

Mac Games: Interesting Times

“**M**ay you live in interesting times” is a statement often attributed to an ancient

Chinese curse, and is a pretty accurate description of the Mac game market today. While there is ample opportunity for explosive growth on the Mac, there are also a few potential potholes for game developers traveling down the Bondi-blue road.

From a technical standpoint, the Macintosh platform is probably in the best shape it has ever been. A decent OpenGL implementation, a growing selection of 3D accelerators (ATI, 3dfx, and potentially Nvidia), and some major game engines already ported (particularly UNREAL TOURNAMENT and QUAKE 3) form a strong foundation for bringing top games to the Mac. The availability of cross-platform game engines is the single most important technology for Mac game developers. Developing a Mac version of an UNREAL- or QUAKE 3-based game is considerably simpler than porting or co-developing a game with a proprietary engine.

But in the middle of this rosy technical picture comes one of those “interesting” wrinkles: Mac OS X. Apple’s big push to build a truly modern OS presents game developers with some difficult choices. While the technical advantages of OS X are numerous (preemptive threads, protected memory, tighter OpenGL integration, better virtual memory, and so on), it also presents a completely different code path for game developers compared to OS 8/9. And for the majority of developers who won’t be able to ship OS X-only games for quite a while, waiting for the consumer market to transition from OS 8/9 to X might be painful. Supporting two distinctly different

Mac operating systems will require some hard work and lots of testing. After the transition, when developers can target OS X exclusively, the Mac should be a technology platform on par with the best.

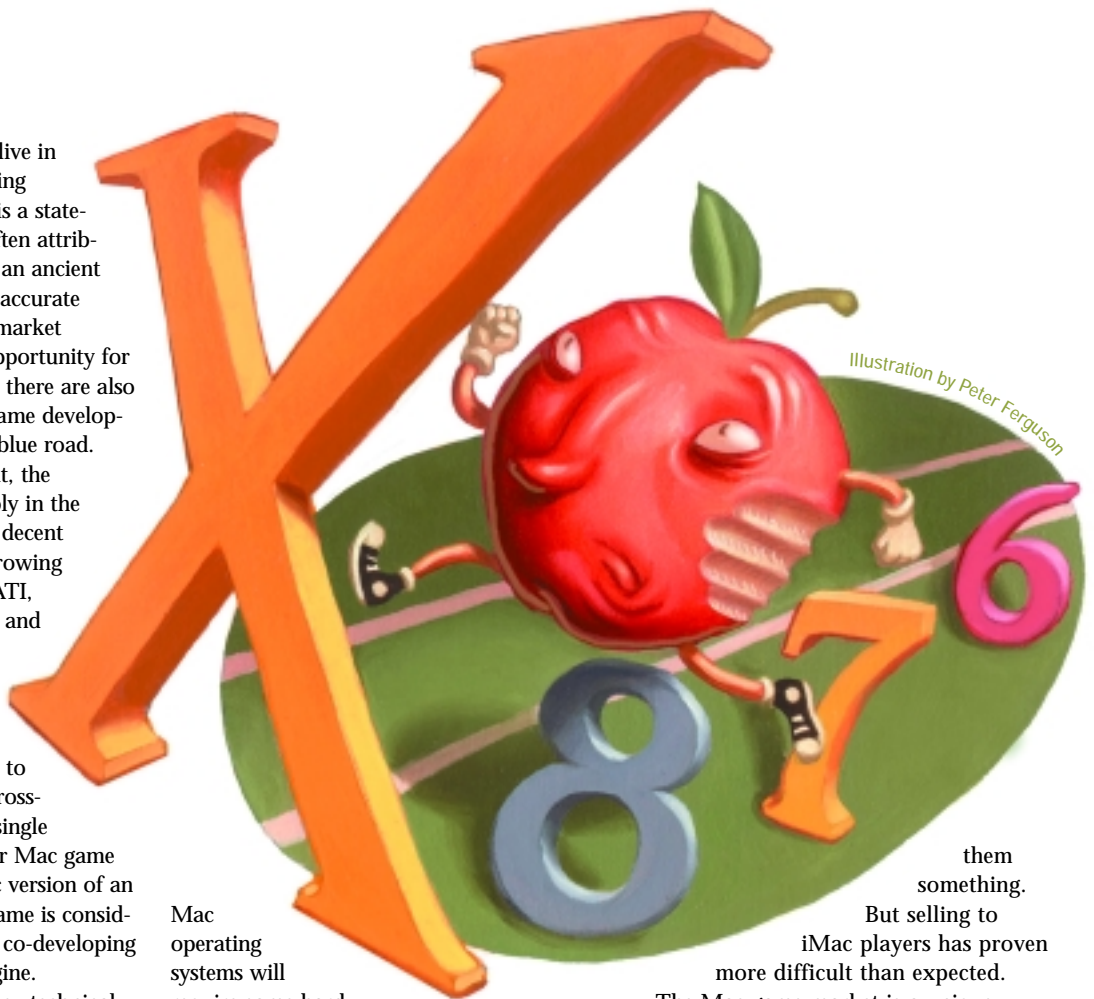
The other half of the Mac gaming equation is the state of the gaming market. For the past two years, Apple has aggressively (and successfully) jumped back into the consumer market with the iMac. However, the market for Mac games has not grown at the same rate as the rest of the consumer Mac market. Ordinarily this would be a problem publishers would love to have: a huge untapped group of potential game players just waiting for someone to sell

them something.

But selling to iMac players has proven more difficult than expected.

The Mac game market is a unique niche. Mac owners seem to be a bit older than their Windows counterparts, and somewhat more selective when buying games. Because many iMac owners are first-time computer buyers, they may be less experienced about finding and purchasing new games. Games featuring a broad appeal and a low price may have more potential to sell to these novice gamers. And while it has been somewhat difficult to sell relatively hardcore games to iMac users in the recent past, there is a good potential for an increase in iMac players moving up to mid-range and high-

continued on page 55



continued from page 56

end titles. In fact, in the last few months, sales figures for the Mac versions of *THE SIMS*, *DIABLO II*, and *DEUS EX* point to a potential breakthrough into this untapped user base.

If this trend continues, Mac gaming can become a profitable area of growth for the game industry. However, the uniqueness of this market will still require specialized knowledge of distribution and marketing to promote good sales. Retail distribution on the Mac has been focused on only a few channels in the recent past (CompUSA, for instance), so it has been important to take advantage of alternative channels such as web sales and catalogs. This, in turn, makes it important to market creatively to Mac users and educate them on their most convenient points of purchase.

Apple has been making strides in increasing the Mac's presence in traditional game retailers, like Electronics Boutique and Babbages, but the alternate means of distribution still remain important.

The final interesting twist impacting the Mac game market is not unique to the Mac. Just as Windows game developers are unsure of the potential effect of next-generation consoles like Playstation 2 and Xbox on their market, Mac game developers are similarly concerned. If, as predicted by some, consoles capture a significant percentage of the PC game market, the Mac will also be impacted.

Cross-platform development will become very important if the game industry

becomes more fragmented due to the next-generation consoles. If market shares diminish for individual platforms, producing games for multiple platforms will be a vital way to maintain total revenue for a given title. Just as in the early 1980s when the market was split between many competing platforms (Apple II, IBM PC, Commodore 64, Atari), game developers producing original content may have to release on as many operating systems as possible to recoup their investment.

The Mac game market continues to have tremendous potential, and a few difficult hurdles ahead. No matter what the future brings, though, it will definitely be "interesting times." 🎮

Mark Adams | Mark is the president of Westlake Interactive, and has been writing Mac games for 15 years. Mark can be reached at madams@westlakeinteractive.com.
