



GAME DEVELOPER MAGAZINE

JANUARY 2001



GAME PLAN

LETTER FROM THE EDITOR

On Game Engines

An interesting phenomenon has been occurring in our Postmortem column recently. Have you noticed it? Our three most recent Postmortem titles have all used licensed game engines. November's DEUS EX used the UNREAL engine, December's HEAVY METAL: F.A.K.K. 2 used QUAKE 3 technology, and this month's STAR TREK: VOYAGER — ELITE FORCE also used QUAKE 3. All three of these Postmortems noted that using licensed technology was one of the things that went right.

Is licensing a game engine right for your title? That depends on a lot of factors. The first thing to examine is the genre of your game. If you're creating a PC-based first- or third-person action/puzzle/shooter title, you have many options available. For sports games, isometric-view games, or flying games, you'll find fewer adequate game engines, but they're still out there. Console games have fewer possibilities still.

Why Buy?

Selling out the bucks for a game engine is not for the faint of heart. There are many different types of licensing agreements, from royalties-only to flat-fee and everything in between. In general, the cost outlay you can expect averages about \$200,000.

The typical game development process takes 18 months and employs 10 people. At a rough cost of \$100,000 per person per year, that game would cost \$1.5 million to develop. If buying an engine would save you two person-years, that would cover the \$200,000 cost.

Unfortunately, none of our Postmortem titles really seemed to save time overall by purchasing an engine. The development process took about the same amount of time that they had expected it would without purchasing an engine. Each development team spent time learning to use the engine, and then augmented it with proprietary extensions such as scripting languages, particle systems, lip-synch systems, and so on. When you're examining game engines, make sure you have planned out the features you're going to need so you

know what systems you'll have to create yourselves. Making alterations to a licensed game engine to add your own features requires quite a bit of time examining the engine code.

One resource that all of our Postmortem titles saved by licensing technology was the number of bodies. For example, DEUS EX only required three programmers. Writing a game engine is a huge task, and not having that task to tackle meant that they could get by with fewer programmers. This is definitely a cost savings. Does it add up to two person-years? Probably.

Other Lessons Learned

By licensing an engine and its tools at the beginning of a project, your artists can immediately begin working on content for the game. If you were writing your own engine, there would be a period of time while the programmers worked on the engine that the artists would be mostly spinning their wheels. In fact, you probably wouldn't want to have many artists on the project at the early stages. When you buy your engine, the programmers have more time to create subsystems and tools which assist the artists and designers in making the game come to life.

Being on the cutting edge by getting a pre-release version of an engine means that every few months you'll have to upgrade your technology. You may have to redesign augmentations you've built on top of the engine, or modify custom tools you've designed. The other choice is to buy a proven stable engine and make your game distinctive through your programmers' augmentations.

By not having to build your game's engine, you have more freedom to concentrate on your design. It appears from our last three Postmortems that that alone is really what makes a game stand out.



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 thyrd@cmp.com

Production Assistance
Audrey Welch awelch@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, Western Region, Silicon Valley & Asia
Jennifer Orvik jorvik@cmp.com t: 415.947.6217
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 magazine is BPA approved

CIRCULATION

Group Circulation Director Kathy Henry
Director of Audience Development Henry Fung
Circulation Manager Ron Escobar
Circulation Assistant Ian Hay
Newsstand Analyst Pam Santoro

SUBSCRIPTION SERVICES

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

INTERNATIONAL LICENSING INFORMATION

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

CMP MEDIA MANAGEMENT

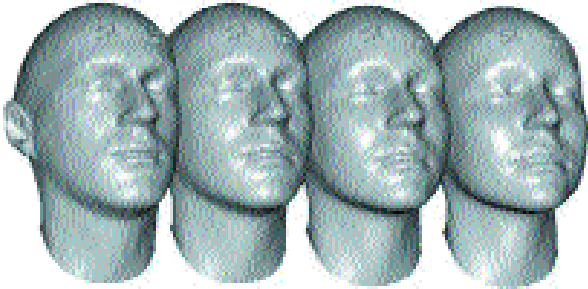
President & CEO Gary Marshall
Corporate President/COO John Russell
CFO John Day
Group President, Business Technology Group Adam K. Marder
Group President, Specialized Technologies Group Regina Starr Ridley
Group President, Channel Group Pam Watkins
Group President, Electronics Group Steve Weitzner
Senior 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





FRONT LINE TOOLS

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



FaceGen SDK morphing a male face into a female face.

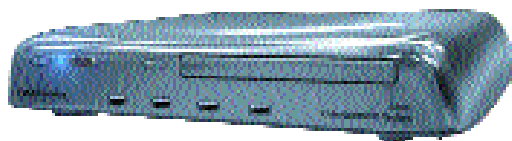
FACEGEN SDK

Singular Inversions has released the FaceGen SDK, a program designed to allow game players the ability to create unique, photorealistic faces easily. Designed to be incorporated into games, FaceGen allows users to modify faces, or create random photorealistic faces for computer-controlled characters. Users can start with any model head, including their own, and use the SDK to warp their shape to any face imaginable. FaceGen also allows for the use of existing morph states or skeletal animations, and existing textures and texture animations. SDK licenses are royalty-free, with prices starting at \$25,000.

FACEGEN SDK | Singular Inversions | www.singularinversions.com

XBOX PROTOTYPE KIT

Microsoft is making it easier for independent developers to make games for Xbox. The company is making an Xbox Prototype Kit (XPK) available free of charge. The kit contains software and information designed to help developers create prototype Xbox games using familiar tools on standard PCs. Those lusting for a complete Xbox dev kit can apply to the Xbox Incubator Program, available to developers who are ready to commit resources to an Xbox title but lack a publisher. Participants in the Xbox Incubator Program submit complete writ-



Codewarrior tools are being developed for Indrema's Linux-based console.

ten descriptions of their game concepts for evaluation by Microsoft's Xbox team. Those invited to join the program can license an Xbox Development Kit for six months to create a prototype and secure publishing and distribution deals.

XBOX PROTOTYPE KIT | Microsoft | www.xbox.com

BIG MOVES FROM MOTEK

Motek and Delft Motion Analysis have collaborated on a new motion capture system that will allow for a capture area of over 900 square feet — more than three times larger than anything previously available. The new system is designed as a solution for capturing big moves in sports or dances that previously had to be broken into smaller pieces of actions and then spliced together.

LARGE-SCALE MOCAP SYSTEM | Motek | www.motek.com

CODEWARRIOR FOR INDREMA

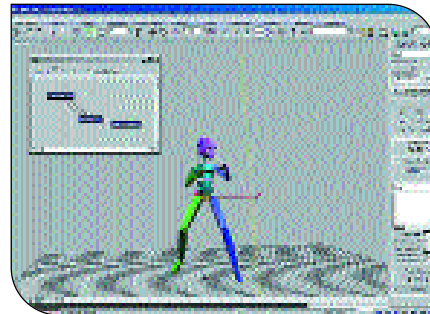
Indrema Entertainment Systems, creator of the Linux-based Indrema game console, and development tool maker Metrowerks, have agreed to jointly produce a version of Codewarrior for Indrema's console, which will be included for free or at a nominal cost as part of the Indrema Entertainment Software Development Kit (IESDK).

Other key technologies in the IESDK will come from Metro Link, a maker of graphical display software for Linux systems, including Open Stream for video and Xtrema for display management. The IESDK will include several open standards for game development, including Mesa3D for 3D graphics and OpenAL for 3D audio. The IESDK is available for download at the Indrema Developer Network web site.

CODEWARRIOR TOOLS FOR INDREMA | Indrema Entertainment Systems | idn.indrema.com

DISCREET SHIPS CHARACTER STUDIO 3

Discreet is shipping Character Studio 3, the newest version of its character animation extension for 3D Studio Max. Character Studio 3 adds new crowd and behavioral animation tools, as well as accelerated physique skinning performance and improved inverse kinematics tools. The Crowd toolset



A motion flow screenshot from Discreet's Character Studio 3.

allows members of a crowd scene to be assigned behaviors such as wandering, seeking goals, avoiding obstacles, following surfaces, or behaviors from custom Maxscripts, while new animation tools allow characters to be motivated by either hand-designed animations, motion capture, or a mix of both. Character Studio 3 also includes many workflow enhancements, including improved nonlinear animation and track operations. Character Studio 3 is priced at \$1,495, with upgrades from Character Studio 2 available for \$495.

CHARACTER STUDIO 3 | Discreet | www.discreet.com



INDUSTRY WATCH

THE BUZZ ABOUT THE GAME BIZ | daniel huebner

Sega shifts gears. Sega plans to address its recent financial woes by shifting its focus from hardware to software. The company is taking the wraps off an ambitious plan to boost its share of the videogame software market to 25 percent. Sega currently holds a 4.2 percent share; Nintendo leads the industry with a 19.6 percent share. Sega's plans to reach their goal include providing software for rival consoles and licensing Dreamcast technology to makers of PCs and cell phones. "We aim to win the top share of the world market in the near future by increasing the number of platforms which can operate Sega software," explained Sega's strategic counsel Tetsu Kayama. "Our focus on content provision is back in place. Sega aims to become a real game creator." To that end, Sega is looking to overhaul its software development operation, including redirecting more than \$200 million in investment funds from third parties back to its own internal developers. Sega hopes these moves will put the company back into the black as soon as this year.

Winners and losers. Game publishers were anxiously awaiting the launch of Sony's Playstation 2, hoping that the new console would signal the end of an extended stretch of soft game sales. The final fiscal reporting period before the momentous occasion brought decidedly mixed results.

Electronic Arts posted a loss in its second quarter, but it was smaller than expected. Second-quarter revenues came in at \$219.9 million, a significant dip from revenues of \$338.9 million in the same period last year. Consolidated net losses totaled \$35 million, compared with a profit of slightly more than \$20 million in the second quarter a year ago.

Activision fared better, beating its second-quarter earnings expectations. The company reported net income for the second quarter of \$4.3 million, or 17 cents per diluted share, up from income a year ago of \$1.06 million or 4 cents per share. Revenues rose to \$144.4 million, up from \$115.4 million in the same period last year.

Interplay Entertainment may be the biggest winner of the group as the company managed to return to profitability in its



The release of BioWare's **BALDUR'S GATE 2: SHADOWS OF AMN** helped Interplay return to profitability in its third quarter.

third quarter. Interplay reported \$31.6 million in revenue for the quarter ended September 30, a 34 percent increase from revenues of \$23.6 million in the third quarter last year. Net income nudged its way to a bit more than \$100,000, or break even per share, versus last year's \$17 million third-quarter loss.

THQ also reported quarterly earnings that topped forecasts, but fell below last year's results. The company had revenues of \$53.3 million in the third quarter, up from \$44.3 million in the same period last year, with net income reaching \$2.4 million. Last year's third-quarter earnings were \$4.7 million. THQ's third-quarter earnings exclude a one-time charge of \$5.9 million related to the company's purchase of Volition.

The 3DO Company's second-quarter revenues were up, but so were losses. The company reported revenues of \$22.9 million for the second quarter, an increase of 11 percent from last year's second-quarter revenues of \$20.7 million. Net losses reached \$15.2 million, compared to second-quarter net losses of \$5.9 million in the same time the previous year.

Midway Games may have had the toughest time in the final pre-PS2 quarter, taking a significant revenue hit. The company reported that revenues for the first quarter of fiscal 2001 fell nearly 56 percent to \$47.3 million, versus last year's first-quarter revenues of \$106.6 million. That decline in revenue translated to a loss of \$9.9 million, compared to net income of \$11.3 million in the same period one year ago. Midway's rough

financial patch has spurred rumors of a buyout. Viacom chairman Sumner Redstone, who already holds a 28 percent stake in Midway, was expected to take advantage of the company's weak stock price to increase his holdings.

PS2 tariff debate. Sony made an appeal to the United Kingdom to reclassify the Playstation 2 for tax purposes. Sony would like to see the machine classified as a computer, thus avoiding a 2.2 percent levy imposed on products classified as videogames. Customs officials denied an earlier request from Sony to con-

sider the console a computer on the grounds that the Playstation 2 was not significantly different from the original Playstation, which was classified as a videogame. Sony could take their defense on the matter directly to the World Trade Organization if its appeal in the United Kingdom is rejected. ☞



UPCOMING EVENTS CALENDAR

AMERICAN INTERNATIONAL

TOY FAIR

JACOB K. JAVITS CONVENTION CENTER

New York, N.Y.

February 11–15, 2001

Cost: none

www.toy-tma.com/AITF

MILIA 2001

PALAIS DES FESTIVALS

Cannes, France

February 11–14, 2001

Cost: (expo only) approx. \$275

www.milia.com

TED II

MONTEREY CONFERENCE CENTER

Monterey, Calif.

February 21–24, 2001

Cost: \$3,000

www.ted.com/tedxi.html



State Machine a.k.a. (Non) Deterministic Finite State Machine, Finite State Automata, Flow Chart

Help! Zack's in Spain with no Internet connection, so I'm using this opportunity to ask for opinions on an important game programming pattern.

Problem

State Machines are one of the fundamental building blocks of all programs, but they are particularly important in game programming. AIs, puzzles, conversation engines, physical controllers, network sub-systems, and sometimes even programmable graphics shaders use State Machines to handle dynamically changing situations in the game. Put simply, a State Machine is a piece of code that manages the current state of a system, and the transitions to new states based on events. The problem is how to represent a State Machine in a program.

Solutions

Because State Machines are so fundamental to computer science, a lot of thought has been put into creating and analyzing them. There are particularly elegant diagrammatic representations of state machines available. Everyone's seen the intuitive drawings with circles representing states and arrows between them representing state transitions due to events. The diagrams might include actions that occur either during a state (Moore machines) or during transitions (Mealy machines). Sometimes a state will have another State Machine nested inside it.

However, once you've drawn your state diagram, translating it into code is problematic and rarely elegant, in my opinion. There are three main ways to implement State Machines.

Transition Tables. I think this is the most concise of the three, however it is also the most restrictive on the types of events and actions your State Machine can support. An array indexed by states and events is created, and then transitions are looked up in this table. Usually the

table is generated at compile time by a separate program that parses a static description of the specific State Machine, or a compile phase generates the table at run time. You get code like this:

```
State = Trans[State][Event];
```

It's difficult to integrate arbitrary actions or transition events into this scheme. It works well when the types of the events and actions are compatible with array storage (characters in a string search, for example). The table, and therefore the State Machine, is almost completely indecipherable if you're looking at it in a debugger or editor.

Switch Statements. This is the most common expression of a State Machine in C code. You have a state variable on which you switch, and the case statements handle the various states, actions, and tests for events.

```
switch (State) {
    case LANDING:
        if (FootTouches)
            State = LANDED;
            FootForce = 3.4;
    }
    case FLYING: ...
```

This quickly becomes a huge mess. My biggest criticism of this technique is that it blurs the concepts of state and transition together. The case statements are the states, but they also implement the transitions by checking the events and setting the new state, and they often implement the actions for the new state. So, in the example above, the `FootForce` for the `LANDED` state is set in the `LANDING` state, which is nonintuitive. Code duplication when there are multiple paths to the same state is another problem with this method.

State Objects. This is the "new" way to implement State Machines, and is recommended by most books on patterns. Here,

states become objects that know how to react to events, and a central State Machine object manages the state objects. Although this solves some of the nonintuitive control flow problems of the Switch Statement, it's common for a 20-line Switch Statement State Machine to become 200 lines of C++ class declarations and code spread over five or ten source files. This doesn't seem like a win to me.

Issues

The whole point of this pattern is that all the solutions known to me have major problems. We have such an elegant and intuitive way to represent a State Machine as a diagram on a piece of paper, but we don't have the equivalent clarity for the pattern in code. Do you have a good way to implement State Machines? Do you disagree with my analysis above? Is there a language or programming style that gets close to the elegance of the state diagram? Visit the Game Programming Patterns page at www.gamasutra.com/patterns and let me know!

References

UML statecharts are the currently hip way to draw state diagrams. Here are a couple of articles about State Machines with equally bad solutions (in my opinion, of course):
www.cuj.com/archive/1805/feature2.html
www.embedded.com/2000/0008/0008feat1.html

We Want to Hear From You!

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



the 2000
**front line
awards**

W

elcome to *Game Developer's* 2000 Front Line Awards! The Front Line Awards are our

annual opportunity to recognize the most outstanding innovative tools for game development. This year's awards actually cover about 18 months of time since our previous awards issue in June 1999. As you might expect, a lot of fabulous tools have come out since then.

There are so many tools available for game developers, and many of them are must-have packages that you've used for years. But what the Front Line Awards are about is rewarding innovation. This puts all packages on a level playing field — what you'll find in the following pages are tools that provide you, the developer, with new features or techniques that make your lives easier or open up new possibilities in your games.

Tool Evolution

The Front Line Awards also provide us with an annual opportunity to survey what makes a tool valuable. One very clear evolution of the tools during this past 18 months is an emphasis on customization. As developers, you have to be on the cutting edge in order to make your game look better, sound better, and be more fun than the game sitting next to yours on the shelf. No matter how innovative a tool is, you need to be able to modify it to mate properly with your engine, take advantage of the features of your target platform, or enable your artists to work more efficiently.

Another significant development during this period has been the rise of middleware. Since Sony began their middleware program, many new tools and libraries for game development have sprung up. They haven't just sprung up for the Playstation 2, either: Sony's middleware announcement inspired an entire wave of "middleware" that spans all platforms. This gives you many more choices, but also makes it vital for you to have a source that distinguishes the tools that

smell like roses from those that just smell.

Over time, game development has become increasingly specialized. In the Atari 2600 days, the programmer also designed the artwork and music. Now we have programmers, texture artists, character modelers, animators, motion-capture technicians, musicians, sound effects engineers, level designers, and many other increasingly specialized

roles. This year we decided to incorporate several new awards — for game engines and level design tools — to highlight two of these newer areas for which innovative tools are being created.

The Process

This year we sought out more developer participation in the awards than ever before. Nominations were solicited from the development community, votes were cast on a public web site (with appropriate screening of the voters and results), and the winners were determined by high vote count. From this process, eight products of the past year have been singled out as exceptionally innovative, along with our annual Hall of Fame inductee, a special award which honors truly indispensable products that have stood the test of time by loyally serving game developers for years.

We're looking at possible models for the future of the Front Line Awards, as our industry grows ever larger and more diverse. The Scientific and Technical Awards that are given by the Academy of Motion Picture Arts and Sciences could be a good model for us to base the Front Line Awards on in the future. We'll be looking at it, and I invite you to look at it as well and give us your feedback. Check out the design at www.oscars.org/scitech/index.html.

Acknowledgements

Finally, I want to thank everyone who nominated products for this year's awards, voted for the products, or wrote the descriptions that accompany each of this year's winners — thank you so much for contributing your time and effort.

Hall of Fame Award Gimpel | PC-Lint

What do you do when the boss calls you in to fix a bug in somebody else's code, and you have to fix it now? I don't mean today, or by lunchtime, or by midnight. I mean now, right this instant, because the courier is idling in the parking lot waiting to take the gold master to the presser so the game will be on the shelves by tomorrow. Find the bug that crashes the system and you're the hero — you might even get that bonus they forgot to give you last year. Blow it and you'll never work in this town again.

Last time that happened to me I whipped out my copy of PC-Lint, from Gimpel Software (www.gimpel.com). This little gem is a source code analyzer that finds everything that is wrong with your code. "Everything" is not an exaggeration. In the above example, it found the assignment operator that was written where a comparison was intended, the kind of thing that gets harder to find the harder you look. PC-Lint found that bug in 250,000 lines of code spread out in almost 100 source files. All right, so it took 20 minutes to find the bug, 30 minutes to figure out a kludge, and two hours to recompile the whole thing. Not exactly what the VP of development meant by "now," but it was close enough to save my hide.

PC-Lint is the samurai sword of development tools. It'll slice, dice, hack, cut, shred, mince, and chop anything, anywhere. But like a real samurai sword, it's not a tool to place in the hands of an apprentice. PC-Lint assumes you understand C++ completely. Goof up an access specifier and PC-Lint will flag the conflict in excruciating detail. But you better know what an access specifier is, because PC-Lint is a big-league tool. It won't hold your hand and explain it to you.

PC-Lint flags almost 2,000 types of problems in your source code, from simple little things such as incorrect indentation to the nastiest tangle of obfuscated code. Each type of problem can be customized with a large selection of reporting flags. You can prevent flagging a type of error in a file, a single line of code, or a group of lines. PC-Lint works with almost every compiler you've ever heard of, from Aztec to Zortec,

and with all versions thereof, and integrates with almost every IDE out there.

Value tracking of automatic variables? Yep, PC-Lint does that. Detection of unnecessary objects in headers, or unused headers? Yep, that too. Debugging of weak definials? Yep. Every time you stumble across a bug and mutter a curse, wondering why your compiler didn't catch it, PC-Lint would have caught it for you. Everything you've always wished your compiler would do for you, PC-Lint will do in several different flavors.

As a time-saving tool, nothing beats PC-Lint. If you're hunting for bugs, or trying to improve the quality of code to prevent bugs from happening in the first place, PC-Lint is the tool to have. It not only does its job well, it does it far better than any other tool in its class. With bugs consuming so much time in game development, going without PC-Lint is a slow form of suicide. If you don't have a copy of it, fire one of your programmers and get PC-Lint instead. You'll finish your game sooner and it'll run better.

— Mike Kelleghan

2D Art Package Adobe | Photoshop 6.0

Photoshop 6.0 is the tool of choice for 2D work. It is hard to recall how I ever got by prior to the introduction of layers in Photoshop; it was a milestone in innovation. For texture creation and editing as well as layout, Photoshop is unmatched primarily due to its powerful implementation of layers. Version 6.0, while not adding any new features quite of that magnitude, does provide many improvements in managing layers as well as significant interface enhancements and new tools.

Working with complex files that contain many layers is arguably Photoshop's

greatest strength, and now managing those layers has become much simpler with the ability to organize layers into sets. The Layer Set feature allows you to group layers in a folder and perform

operations on them as a group, so you can apply a mask or hide a whole set of layers very quickly. This grouping of layers also makes the Layers palette much easier to navigate because similar elements of an image can now be visually grouped in the Layers palette.

Photoshop's interface is now very customizable with more

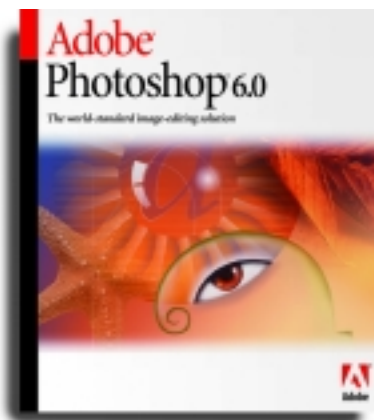
freedom to dock palettes and organize tools. A new context-sensitive Options bar has more options available and the ability to store docked palettes. Palettes docked on the Options bar are easy to access and they free up screen space.

Vector-based graphics have been integrated into Photoshop, so now you can create resolution-independent graphics and masks. Pathfinder operations just like those in Illustrator are now available as well. Vector-based shapes can be stored in libraries for easy reuse.

New text features have been added, such as editing text directly in the image window and the ability to warp type. Another nice feature is the Liquefy command, which allows you to distort a layer or image by painting with brushlike tools.

Photoshop 6.0 includes many improvements that increase functionality, ease of use, and productivity, making an indispensable piece of software for the game artist even better.

— George Simmons



3D Art Package Discreet | 3D Studio Max 3.1

Over the years, 3D Studio Max has become the standard for making real-time 3D models for videogames, and for good reason. Release 3 is a powerful polygonal modeler, has an easy-to-use material editor, contains great mapping tools, and has strong plug-in support, mainly due to its popularity. It is also priced reasonably when compared to other high-end 3D software.

Max isn't the strongest 3D utility for all applications right out of the box, but when it comes to modeling real-time models for games, it's very hard to beat for one important reason: simplicity. Every tool needed for modeling is right at your fingertips. An intuitive user interface makes things even easier, and Max 3's interface is completely customizable. Users can create tabs that contain any favorite tools and commands to fit that user's individual needs.

Creating and adjusting materials takes no time at all. Tools such as Unwrap UVW and Surface Mapper make mapping coordinates extremely adjustable and easy to apply to objects with irregular shapes. Max has an enormous plug-

in library, both freeware and retail. Many powerful plug-ins are available, and more are on the way every day because of the popularity Max has built over the years. Many plug-ins that were considered "must-haves" for release 2.5 now come standard in release 3, making Max a fantastic value.

Max also comes with a free SDK that includes more than half of the source code, making it easy for programmers to create custom tools, exporters, and plug-ins to fit any developer's needs. When it comes to polygonal modeling, Max is simply the best choice out there. The simplicity, capabilities, extensibility, and price have helped make Max the best-selling 3D software in the game industry.

— Chris George



Art Utility or Plug-in ACDSystems | ACDSee 3.1

Chances are, you've got somewhere around 40 bazillion images in your current game. Whether it's interface screens, textures, sprites, promotional artwork, conceptual drawings, raw scans, or any other image related to your game, you need a way to keep track of all this stuff. ACDSee

makes this daunting task much easier.

Over the past few years, ACDSee has grown from a simple graphics file viewing utility to a full-blown suite of image display, organization, conversion, and editing tools. As a result of this evolution, ACDSee has become the image viewer of choice among computer enthusiasts and game developers alike.

ACDSee's blazing-fast renderer ensures that images can be viewed instantly. By using the convenient browsing feature, you can quickly scan through your graphics data accompanied by an automatic thumbnail view. Files can be viewed directly from compressed ZIP and LZH archives. Comprehensive file listings are available with the click of a button. Also bearing mention are the comprehensive search functionality and image annotation capability.

The latest edition of ACDSee, version 3.1, attempts to provide some of the same services of high-end professional tools but at a shareware price. Among these new enhancements is a batch function that can run a series of simple operations on a selection of files. These basic features are buffeted by an array of plug-ins which provide a growing list of functions, such as generating contact sheets, automatically creating HTML "albums," reading obscure file formats, and various other enhancements to the standard package.

ACDSee is also becoming the tool to manage all media, not just images. The last few versions have included support for audio files in addition to its vast array of supported image and video data. In addi-



ACDSee 3.1

tion, an extensible plug-in architecture guarantees that future formats can be added easily, as well as custom modules for

your own proprietary formats using the ACDSee SDK.

Even if your needs eventually outgrow the comprehensive feature set of ACDSee, it's worth exploring this low-cost option before laying

down the dough for far more expensive and marginally more useful applications.

— Ralph Barbagallo

Programming Tool MicroEdge | Visual SlickEdit 5.0

There are many tools for programmers available on the market, but if you're specifically looking for a text editor for your project, Visual SlickEdit 5.0 stands out above the rest. It's powerful, customizable, and easy to use. I like to program using VI, and before I started using Visual SlickEdit, I would switch between VIM (VI under Windows) and Visual C++ to get the best features of both editors. Well, Visual SlickEdit combines the benefits of both and adds much more, providing accurate emulation of Visual C++, CUA, Brief, Emacs, and VI.

The Context Tagging, which includes Auto List Members, Auto Function Help, and Context Navigation and Context Preview, is superb. Visual SlickEdit maintains a database of functions contained in your source and header files, and indicates the function you are currently editing. Files are automatically retagged when they are added to a project, saved, or an idle

time threshold is reached. Most importantly, the tagging is impressively fast, efficient, and can handle 65,000 files.

Other customizable features that help increase productivity are the C/C++/Java Code Beautifier, Comment/String Spell Checker, Multi-file Search-Replace, and an intuitive macro language with macro recording capabilities. The code beautifier is particularly useful; it reformats your indentation, tag styles and brace styles to suit your preference.

SlickEdit makes extensive use of dialog boxes and windows to allow users to specify their preferences or custom-define macros and key bindings. The editor is so rich with features that you will find yourself discovering new ones even after months of use.

Beyond the editor itself, Visual SlickEdit provides a robust environment for any project. In the console industry, we frequently find ourselves dependent on a variety of awkward environments with specialized compilers, linkers, art tools, and so on. In my work at Nintendo of America, I have found Visual SlickEdit to be very useful in providing a stable, easy-to-use interface for such environments. For example,

Visual SlickEdit can recognize error outputs of several different compilers (including command-line compilers such as gcc), and automatically bring you to the line in the source file where the error occurred. It incorporates very easily with makefile-based projects, and, for PC developers, Visual SlickEdit can open a Visual C++ workspace seamlessly.

Visual SlickEdit also provides tools similar in functionality to Unix's diff, grep, and find commands but with a user-

friendly, easy-to-learn interface. The file-difference tool, named DiffZilla, is sophisticated. The diff dialog can operate on multiple files in a directory tree, while excluding specified directories or file types.

With all these powerful features, Visual SlickEdit will increase your performance and efficiency by providing you with the tools you need to get the job done.

— Dante Treglia II



Programming Library Microsoft | DirectX 8 Direct3D

After almost half a decade of living in OpenGL's shadow, Microsoft's 3D graphics API has come into its own. It's amazing how Microsoft has turned Direct3D from the laughingstock of the game development community to the bleeding edge of high-performance 3D graphics technology.

Back in the Stone Age of Direct3D, we had the horror of execute buffers, sparse documentation, inconsistent driver implementation, and lengthy setup routines that seemed to dwarf the size of the application's main code. However, with the last few releases of Direct3D, almost everything has become streamlined. You can practically set up your entire Direct3D application with a single call. Robust framework source is provided with each release. And the documentation is actually helpful.

Previously, Direct3D was playing a game of catch-up with developers. With DirectX 8, Microsoft is now ahead of the curve. With the release of QUAKE 3, using programmable shaders in real time has become a hot topic. Direct3D now includes built-in support for pixel shaders. It also has more support for vertex blending and skinned models — an important element now that skeletal animation systems are becoming commonplace. These days, if you want to learn about the latest in real-time 3D graphics technology and techniques, a fine place to start is the Direct3D API.

Direct3D is becoming not only a 3D graphics API, but an entire toolset.

The new release is accompanied by handy plug-ins to export model and animation data from such popular packages as Maya and 3D Studio Max. Combined with Microsoft's growing suite of tools for use with other DirectX components, developing games on the Win32 platform is becoming easier every day.

While OpenGL remains mired in bureaucratic processes to include new elements, every year we get a new version of Direct-

3D rife with brand new standardized features, optimizations, and tools. Microsoft has been quick and responsive to developers — and it shows. With extensive developer support on Microsoft's various newsgroups, thorough documentation available on their web site, and an ongoing series of developer conferences, keeping abreast of the increasingly large set of Direct3D features is becoming a bit more manageable.

Sure, all of these Windows API goodies are keeping many game developers somewhat shackled to Microsoft's platform. But if you are looking for cutting-edge Win32 multimedia performance, DirectX 8 could quite possibly be your complete choice.

— *Ralph Barbagallo*

Audio Tool Factor 5 | MusyX

The first time I heard Factor 5's MusyX tool, it was playing a John Williams-like symphony that sounded absolutely brilliant. The music was set up as a MIDI file, just a bunch of scripted notes, but used SMA_L (Sound Macro Language) macros to modify some of the sounds. This gave the scripted music a symphonic quality unlike what you'd

normally hear when a choir of digitized violins all play the same note. In that case, you would typically hear just one note, since the waveform would stack up on itself, but this actually sounded like a real orchestra belting out the *Star Wars* theme.

MusyX is a synthesizer program that runs on your PC but targets audio for Game Boy Color, Game Boy Advance, Nintendo 64, or Gamecube. You can use whatever music composition tool you're used to — Cake-

walk, Cubase, whatever — and have it play to the MusyX slave program which is either running locally on your own PC or remotely on your network (it communicates via

TCP/IP). The MusyX slave can emulate your target platform and play through your PC speakers, or send to your development hardware and play the music on the real thing.

The Game Boy Advance version lets you

mix 32 sampled voices and also do Dolby Surround encoding. (Of course, you'll need to wear your headphones to hear it properly.) The N64 version supports 32 much higher-quality voices and also includes a 3D sound API.

The Sound Macro Language lets you modify notes for music as well as create sound effects. You can do this through the graphical drag-and-drop interface, dragging existing macros to affect your sounds or creating your own macros. There are a ton of different macros you can create — more for the more advanced platforms — and if you really wanted you could take your Game Boy Color music with SMA_L info and play it on the N64 just by changing the slave program and making some adjustments to or replacing the macros.

Not much information is available about the Gamecube version at this point, but if it's as good as the previous versions it should make creating interactive music for Gamecube much easier.

— *Jim Verhaeghe*

Game Engine Epic Games | UNREAL Engine

Here at Ion Storm Austin, we used the UNREAL engine to create DEUS EX, our hybrid RPG/shooter immersive sim, which was in development for over two years, including preproduction. As a development group, we wanted to be content providers, not technology



Screenshot from MusyX



providers, so we were looking for an engine that would let us spend a significant portion of our time working on gameplay. The marriage of our design/development vision and the UNREAL technology worked out very well.

The UNREAL engine technology enabled us to get started quickly, instead of waiting a year or more for our programmers to create an original technology package. Almost as soon as we started, we were building out spaces and getting a feel for what a DEUS EX environment was. The programmers could take some aspect of the design document and prototype it in short order. This allowed us to go through many iterations of game design features rapidly, for instance how our nanotech augmentations allowed the player to interact with the game environment. The technology was incredibly stable (from the perspective of someone familiar with in-house engine technology used at other companies). DEUS EX almost never crashed during development, which again allowed us to focus more time and energy on content creation, instead of fire fighting.

The UNREAL engine package is powerful and well supported, especially from the artist and designer perspectives. The UnrealEd level editor is subtractive, which I prefer greatly to additive editors. We built all the maps and missions for DEUS EX with UnrealEd. Our programmers wrote a stand-alone tool called ConEdit for the character conversations, but in all other cases we used UNREAL technology as a starting point. As mission designers, we loved working with it — people often had side-project maps going, working on, say, a temple from their fantasy RPG campaign, the interior of their apartment, or some small scene from a favorite movie. That says a lot; people saw it as a fun, enabling creativity tool, rather than a horrible beast that had to be struggled with daily. The designers on our team actually enjoyed using UnrealEd.



There were a few rough aspects that required working around, but no technology is perfect. All games are designed around limitations. At key points during development, we came to understand how things could best be done with UNREAL, embracing the technology rather than fighting it.

UNREAL was a great choice for us — it enabled us to make a game we're proud of, and a game that seems to have made a lot of players happy.

— Harvey Smith

Level Editing Tool Valve Software | Worldcraft

Back in the days of WOLFENSTEIN 3D and DOOM, level design software was in a horrible state. Unstable editors, zero documentation, and complex construction techniques made level design for games inaccessible to all but the most hardened developers.

With the advent of DUKE NUKEM came the Build engine and editor, which spawned thousands of amateur level designers who proceeded to fill the web with their creations.

Many people left Build behind to move onto greener, more 3D pastures when QUAKE was released, and so the QUAKE engine level editor was born. A few editors rose to the challenge, QuArK (the QUAKE Army Knife), Radiant, and Worldcraft being the most notable editors of their time.

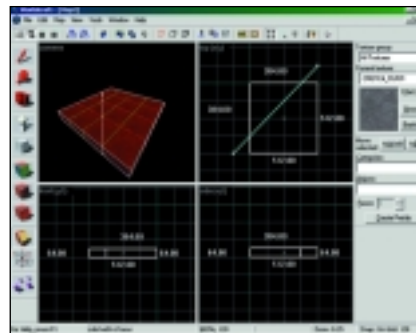
In its previous incarnations, Worldcraft was, quite frankly, not the greatest editor on the block. What it offered in ease of use, it took away in instability, poor 3D display, and resource digestion. When Valve Software picked it up as their in-house level design software, it was still stuck in this state. Now, at version 3.3, it

has progressed in leaps and bounds, adding features users have been clamoring for since the beginning.

First among these new features is a fast, smooth, stable, and very much appreciated OpenGL 3D window which, while requiring a video card capable of running OpenGL in a window, is a great step up from the previous version, and makes a lot of people's lives a lot easier. Version 3.3 also introduces the ability to preserve texture alignment on rotated brushes. Previously, if you wanted to rotate a crate, say, five degrees left, you'd have to manually rotate and position every single texture on every single face — while this was bad enough on a six-sided box, imagine the fun you'd have on anything larger. Worldcraft 3.3 brought us the rotating texture lock, which made creating realistic environments a much less time-consuming process. Gone are the days of uniform objects in a straight line in most levels. Another useful feature is the ability to display actual sprites in the 3D window, rather than a green square. Worldcraft 2 popped up a colored cube every time you placed a sprite — now you see them animated in real time, which is a blessing to people trying to position sprites perfectly.

Worldcraft also provides an amazing amount of support. While it's a bit of a

beast to set up for the first-time user, there's plenty of help available from Valve and on the web. Some people still stick by their old-school guns and slog it out with QuArK or



The Worldcraft level editor

whichever editor is still favored among the level design dinosaurs. Good luck prying Worldcraft 3.3 out of my hands, though. With the continued support from Valve and the fantastic features added in every new update, Worldcraft is one piece of software that's definitely on the up.

— Simon Westlake

Battling Beetles

Now that the new millennium has actually started, it's time to reflect on the year that just ended. I've covered quite a bit of ground in this column: nonphotorealistic rendering, free-form deformation, 3D painting, texture wrapping, and a bunch of other stuff for creating, animating, and rendering 3D objects. It's probably apparent to most of you the types of game development situations where these different techniques are useful; however, I still often get e-mail from readers that basically sound like, "The technique you described in the magazine for doing 3D_GRAPHICS_TECHNIQUE is pretty cool. But, I am doing a CURRENT_GAME_GENRE title and I don't see how I can use it. When do I need that technique?"

Now that my game developer toolbox is all full of nice little pieces of 3D technology, it's time to pull it all together and actually do something with it. We start with the game design. First of all, let's hope I have one. As the technical lead, it's always better to have a fully developed concept before you actually start creating the game. The few times I have been involved with a "design it on the fly" kind of project, it's always ended in disaster. But let me start by assuming that I have a design.

The Design for Battle Beetles

Pitch: "It's a fighting game with a heart, but it's really funny."

Genre 3D fighter similar to Sega's VIRTUA FIGHTER or Midway's READY 2 RUMBLE.

Style Animated-cartoon-style fighting game with cartoon physics (squash and stretch).

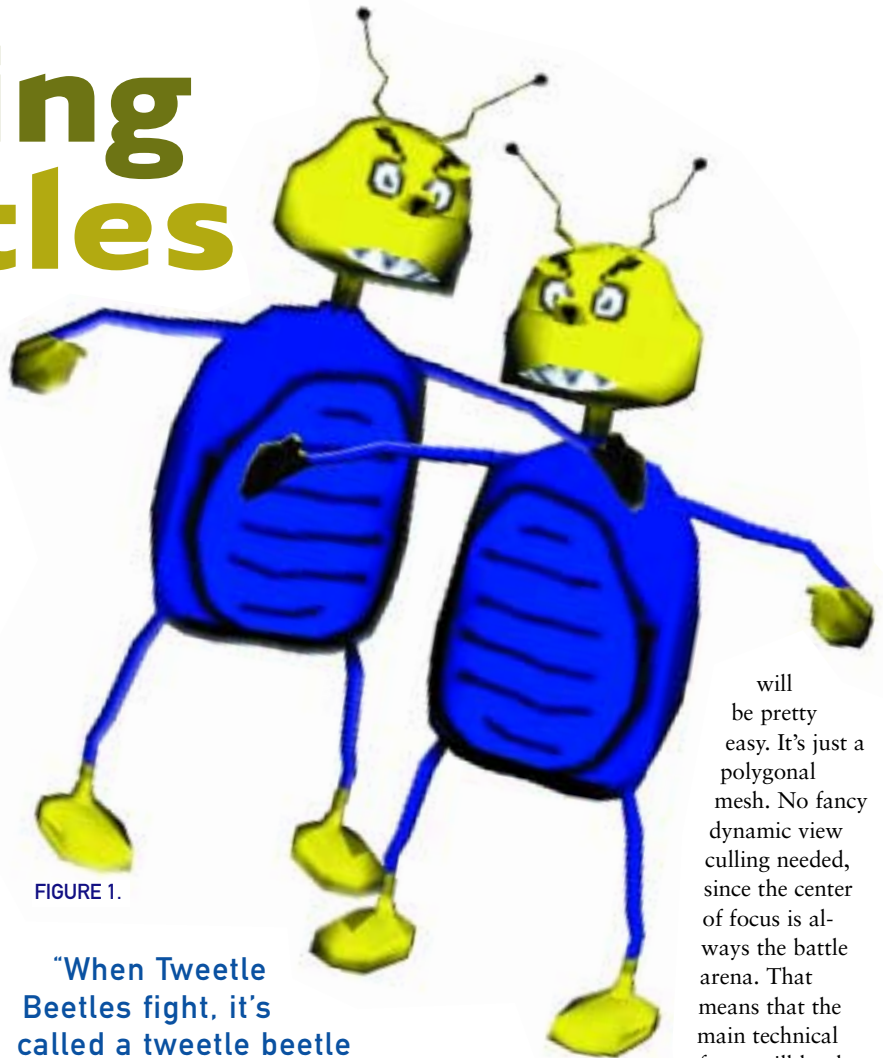


FIGURE 1.

"When Tweetle Beetles fight, it's called a tweetle beetle battle." — Dr. Suess

Levels Various fantastic cartoon areas like the "gooey blue goo puddle" and the "battle in a bottle."

Platform The latest and greatest 3D game platform (meaning polygon count and graphics features are not a major factor).

Target demo Mass market and early adopters (we want the sexy high-end graphics but want to sell a ton of them).

Delivery 18 months from start of production (aren't they all?).

Poor me, that's all I get. From that, I need to create a technical design document. Unfortunately for me, that means I need to decide how a "fighter with a heart" actually plays. The environment that they fight in

will be pretty easy. It's just a polygonal mesh. No fancy dynamic view culling needed, since the center of focus is always the battle arena. That means that the main technical focus will be the characters.

Single-mesh polygonal characters are the obvious choice. I don't need any fancy continuous level of detail scheme, since the players will be full-screen and close. The game style also dictates a fairly simple mesh. No need to go crazy with the polygon count. You can see an example of the beetle characters with simple texture maps battling in Figure 1.

To animate the characters, I could export various mesh key poses and morph between the poses in the way that QUAKE characters do. However, deforming the characters with a skeletal system makes more sense for some key reasons. A contemporary fighting game requires a lot of animated moves. Morphing all these

JEFF LANDER | Presently, Jeff is probably pondering perplexing problems with polygons. Jot Jeff a jaunty jingle at jeffl@darwin3d.com.

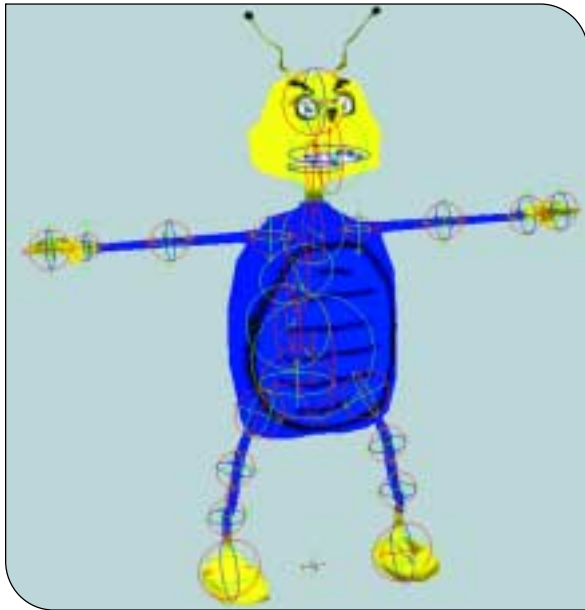


FIGURE 2. A beetle with boundary spheres approximating the shape of the character.

frames would require a great deal of memory. Animating the skeleton will require a great deal less memory. Also, using the skeletal system for animation will allow for the creation of dynamic moves on the fly. Plus, I already have the code prepared to support the matrix deformation needed for the skeletal animation system.

Collision Detection

Handling collisions between the characters will be the most important technical issue of the game. Fighting games depend on good collision detection. I could simply track the character states and distance between them. If one character was in the “punch” pose and was close enough to the other player, I could call it a “hit” and make the characters react. This was typically the way fighting was handled in the 2D fighting games of the recent past. However, for the modern 3D fighting game, that is no longer adequate. Players like to see realistic reactions. If one player hits another in the head, the head should snap back. Likewise, a punch that is blocked should not continue through the body of the character. This is a key problem with many fighting games. They either cannot

detect the collision correctly or lack the ability to have the characters react. You often see little puffs of magic dust or sparks to cover the fact that the hand of one character passed through the torso of the other.

To get accurate collision detection for the characters, or at least accurate to a point, I need to check if body parts are actually in contact. A single boundary for the entire character will not be good enough. I could go to the level of checking for collisions between individual polygons, but that would be more detail than I actually need. One method for more detailed collision detection is to use

a bounding sphere tree. In this technique, boundary spheres are arranged in a hierarchy where each level of the tree bounds a more detailed representation of the object. For example, the root of the boundary tree would be a sphere that encloses the entire character. The next level has boundary spheres for each arm, each leg, the head, and the torso. The branching could continue until each individual polygon is bounded by a sphere.

This again is probably more detail than I need. However, it’s a good starting point. I have a character made of a mesh that contains a skeleton composed of bones that roughly follow the shape of the character. I can add a floating point `bsphere` value to the bone data structure that represents the radius of a bounding sphere at that bone root location. This will give me a series of boundary spheres that approximate the shape of the character. As a side benefit, since the boundary spheres are “attached” to the bones, they follow the animation of the characters. You can see the boundary spheres defining the character in Figure 2.

The way I defined these sphere sizes and positions, they don’t entirely cover the object. For example, it may be possible to pass an object through the character without hitting a boundary sphere;

for example, between the elbow and wrist is a good example. I could close the gap by adding more spheres, for example, between the bone roots or by using oriented bounding boxes to define the object instead of spheres. However, these spheres are good enough for my fighting game application.

This same idea is useful for other game genres as well. For example, one of the features that players liked about Raven Software’s *SOLDIER OF FORTUNE* was the ability to selectively hit regions of the character’s body. This feature could easily be accomplished using the sphere-tree approach.

Applying the Skin

So I have a polygonal character with an embedded skeleton and boundary spheres attached. However, unless I attach the skin to the skeleton, the beetle is going to have a tough time moving. As I have mentioned before in this column (“Over My Dead, Polygonal Body,” October 1999), vertex weights are used to associate each vertex with the bone matrix in the skeleton. However, generating these weights is as much art as technology. If you are fortunate enough to use an advanced 3D animation system such as Maya or 3D Studio Max, you can use these tools to generate the skin weights. Of course, the challenge then becomes getting the program to give you those values, but that problem will have to wait for another article. If you do not have a program like this, you will need to roll your own.

If you realized that the 3D paint system I created for last September’s column (“Art and Intelligence: 3D Painting”) is well suited for the job, you get a cookie. In that program, I already had the capabilities to paint on the surface of an object. Making the system paint vertex weights instead simply involves hooking a few pieces together. However, for a reasonably sized model with a lot of vertices, you are not really going to want to paint each vertex weight by hand. You want to have a starting point. The 3D animation packages use a default mapping to get things going. I want to re-create this functionality to give my program a head start as well.

The obvious starting point would be to say that the vertices are influenced by the bones that are close to them. I need to create a falloff function that describes the amount of influence a bone has on a vertex. When the weight is 1.0, the vertex is under full influence of that bone. When it is 0.0, the bone has no influence at all. I have seen a

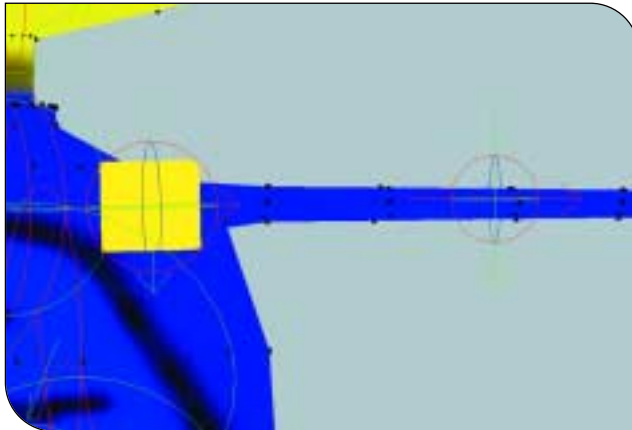


FIGURE 3. Automatic arm weighting.

variety of falloff functions, but for my models the following function gives me the best results:

$$d = Mv$$

This weight is calculated for each vertex-bone pair. The weights for each vertex are then sorted by weight. You can limit the weighting to a certain number of bones by zeroing out the weights beyond that number. The weights for each vertex then need to be normalized so that the sum of the weights for any vertex is equal to 1.

The algorithm for the automatic weighting function is:

- For each Vertex in the mesh
 - For each Bone in the skeleton
 - Calculate the weight using the above formula
- Sort the vertex-bone pairs by weight
- Zero out the weights beyond the desired blend number
- Normalize the weights so that their sum = 1

The tricky bit is deciding on the distance function to use when weighting. The simple method would be to calculate the distance from the vertex to the bone root. However, this is not ideal. Consider the case of the arm in Figure 3. The root is near the shoulder; however, there are vertices all along the bone structure. Using the simple root distance function, the vertices along the arm bone would not be considered.

I need a distance function that considers the distance from any point along the bone. Fortunately, this is similar to a

problem that I ran into long ago. Back in January 1999 (“Crashing into the New Year,” Graphic Content), I described almost this exact scenario for use in making sure that a point is not too close to a wall. I’ll recap the technique here.

What I want to know is how far away the test vertex, v , is from bone segment A . An easy solution would be to find the nearest point, n , to the test point on the bone segment and measure the distance to it. First, I create a vector, B , from the test vertex, v , to bone root b_1 . I can dot this vector with the bone segment A . This will give me the cosine of the interior angle. If this angle is 90 degrees or greater, the nearest point is the root itself and I am done. But let’s say that the dot product gives me 0.7 or the cosine of about 45 degrees. I will then do the same thing on the other side. I create a vector, C , and dot it with the segment A . If it had returned an angle greater than or equal to 90 degrees, point b_2 would be the closest and I would be done again. In this case the dot product returns 0.75, or the cosine of about 40 degrees. Now that I have the two dot products, a linear ratio will solve the problem:

$$\sum_{n=0}^{vertexcount-1} [d_n - Mv_n]^2$$

This new point, n , can then be used to check the distance to the vertex.

In practice, this works much better than the basic root distance. However, I have found that if you use the entire bone length for the test, it interferes with the next bone in the hierarchy. To coun-

MORE ON ARBITRARILY DEFORMING MESHES

In last month’s column (“Pump Up the Volume: 3D Objects That Don’t Deflate”), I was trying to find a way to generate a local coordinate frame for an arbitrarily deforming mesh. The method I described could easily be termed a “hack.” What I did then was identify vertices that roughly define the major axes of the object and then use the average of those points as a local frame. For objects that have vertices roughly along their major axes, things work pretty well. However, for a more complicated object, things are more difficult. Averaging the outer hull vertices also may not always give the best result. After speaking to Chris Hecker and David Eberly among others, it appears that there may be a more mathematically correct, though more computationally expensive solution. If I consider an initial vertex, v , that corresponds to a deformed vertex, d , I want to solve the equation

$$d = Mv$$

where M is the matrix that transforms the vertex to the deformed position. This leads to a problem of error minimization. I can measure the distance from d to Mv for each vertex and attempt to minimize that distance. I want to find a matrix, M , that will return the minimum possible value for the function:

$$\sum_{n=0}^{vertexcount-1} [d_n - Mv_n]^2$$

This matrix will represent both the translation and rotation that caused the object to get into its deformed space. It is complicated a bit by the fact that I want the transformation matrix to be composed of only rotations and translations. A general transformation matrix can also scale the object up or down to cause the deformation. To keep this from happening, I need to add constraints on the problem to keep the scale uniform. I will be investigating this further and reporting the results here in future columns. If you have any ideas on the subject, drop me an e-mail.

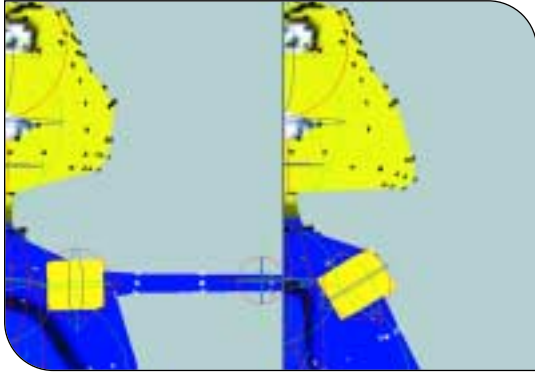


FIGURE 4. Too much influence.

teract this problem, I use 75 percent of the bone length as the test segment for the distance. That seems to give me a reasonable amount of influence without too much interference.

That is not to say that the algorithm is perfect. I am just using a simple distance check. This check assumes no knowledge of the actual mesh topology. So, as you can see in Figure 4, the influence can fall off onto

objects that it should not affect. In this example, the arm rotation is moving vertices in the face.

I could probably combat this problem by considering the distance traveled along the topology of the mesh. But it is easier to clean up these issues by hand using the weight painting function. That is what makes this problem artistic as well as technical. At visual effects companies, there are staff members that do nothing else but attach skeletons to characters and set

up the skin weighting. The game industry is probably not to that point yet, but may soon get there.

I Have Bones but I Can't Walk

I haven't addressed either the animation or rendering aspects of my technical design document yet, but that will have

to wait until next month. Until then, start thinking about how we can use skeletal deformation and animation techniques and still get the cartoon physics type of squash and stretch that we want for the characters.

On *Game Developer's* web site at www.gdmag.com, you will find source code for automatically computing the vertex weights of an object, as well as modifications to the 3D paint program so that you can paint the weights. Thanks to Christine Lander for creating the Battling Beetle model. 🐞

FOR MORE INFORMATION

Seuss, Dr. [Theodor Seuss Giesel]. *Fox in Socks*. New York: Beginner Books, 1965.



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

File Organization and Naming Schemes

Way back (and I mean *way* back) when I art-directed my first game, it was just sort of assumed that I knew what I should call my files and how I would begin to organize my data. I was more or less left to my own devices to figure out just what needed to be done. Well, truth be told, I didn't have a clue. I made it up as I went, and luckily, it was back in the Jurassic period of game development when floppy disks were all the rage. Even when I wasn't well organized, it didn't matter too much since there just weren't that many files to worry about.

Times have changed. CDs have become the standard and DVDs are starting to make a showing in the gaming industry. Mass-media storage capabilities are standard on the next-generation consoles. With the size and scope of most games coming out now, art teams generate thousands upon thousands of files. They range from the working art production files to the final files used on the CD or DVD. There are 3D mesh files, texture files, animation files, GUI bitmaps, special 2D animation files, vector files, support files, and so on. With all of that data floating around, organization becomes critical. Without it, work will be misplaced, lost, and have to be redone. Hair will turn gray and blood pressures will rise as team members try to keep a handle on this ever-expanding issue.

It Could Happen to You

Although much of what I'm going to talk about is rooted in common sense, it still deserves to be explored. Sometimes we get so tied up in the details that we can't see the forest for the trees.

For a real-world example, let's examine

a project I worked on a few years back. I was brought on midway through the development process of a large project. The art team was busily creating art, and had been for well over a year before I joined them. One of my first tasks was to try to get a handle on just what had been created and find out where it was.

It turned out that a formalized method of file naming hadn't been established, or if there had been one, no one was adhering to it very stringently. In addition, the directory structure for the art assets, both working and final, didn't exist in any logical sense. The files were sometimes on individual hard drives, sometimes on network drives, or occasionally backed up onto CDs. Team turnover had exacerbated the problem as well.

Under these circumstances, it took me the better part of a month just to get to the point where I could make a spreadsheet showing which assets were complete, which were incomplete, and where everything was.

Additional problems occurred because of incorrectly named files, which were linked to untold lines of code and scripting files. Correcting them (and there were a lot of them) would have taken too long, so we decided to leave them. These files were a never-ending source of confusion. Logic was tossed out the window, and we had to develop a vast lookup spreadsheet just for tracking the data.

While this is an example of a worst-case scenario, it exemplifies the issues well. With the proper structure and forethought, we could have avoided a large expenditure of art resources.

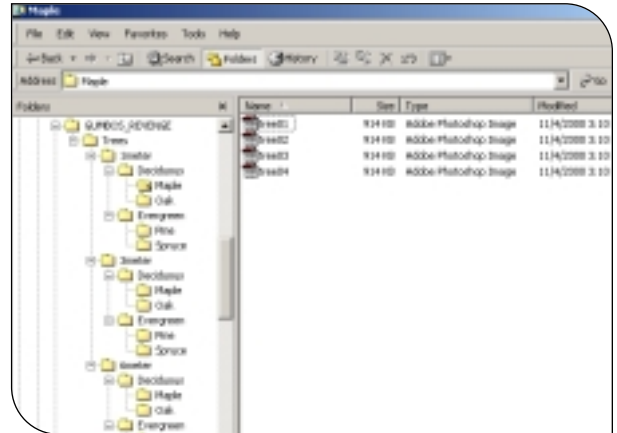


FIGURE 1. A simplified directory structure that is headed down the wrong path.

File Organization

Based on that experience, and refined in the years since, I have compiled some rules to live by: First, files should sort alphabetically and numerically in directories. Second, Excel is your friend. And third, fast, cheap, or good — pick any two.

All right, now that I've gotten that out of my system, I'll explain why I arrived at these rules in a bit more detail. The first rule pertains to the way your data will sort. If you design a directory and file system that is logical and flexible, you will reap the rewards down the road. If you just name files and directories randomly, or as you need them and without forethought, you will oftentimes find yourself painted into a corner. The farther you are into the development cycle of a project, the worse it gets when you have to make sweeping changes to files. Often, other team members are directly affected by your changes. This can include programming, level design, and just about every facet of development.

Let's take a look at an example and walk through some of the problems and

potential solutions: Your project is GUMBO'S REVENGE, a real-time 3D game. It has a 3D world that will be filled to the brim with objects, trees, and all of the necessary graphical splendor to make it a big hit. As you begin to create the art assets for the game, the issue of organization rears its ugly head.

Let's take a standard tree object as a case in point. The initial design calls for your trees to be of varying sizes, including one meter, three meters, and six meters tall. There are also a couple of evergreen and deciduous varieties. One of the first things you'll want to do is to determine what to call your file, and where to put it. In establishing what to call your files and directories, you might ask yourself some questions. What thing is the first determining factor in a directory-based file name? Is it the size? The color? The type? Do I care? (Hint: answer "yes" to the last one.)

So let's say you decide that it goes in the following order: size, general type, specific type, and so on. You determine that there will be a lot of trees in your game, so you decide that the first order of business is to set up your directory structure (see Figure 1).

Now, everything is going along just fine until you decide that you need 12-meter trees. When you add the 12meter directory, it situates itself at the beginning of the other directories, following the standard alphanumeric sorting.

Well, now your directories are out of order. Not a big thing, but a nuisance just the same. So you go back and rename your first three directories 01meter, 03meter, and 06meter. This allows your files to sort correctly, and everything proceeds as it should — for awhile (see Figure 2).

In your first directory (..\1meter\deciduous\maple), you create four trees. Being the logical person you are, they get named: TREE01, TREE02, TREE03, and TREE04. Then, you move on to the oak trees, and you name the first four trees . . . uh . . . TREE01, TREE02, TREE03, and TREE04. Your logic is that they are in different directories, so they can share the same name without any problems.

After a couple more directories get filled with files of identical names, you realize that you may be heading down the wrong path. Your first clue occurs when the level design-

ers start making little dolls that look like you and pinning them to their cube walls. And they use way more pushpins than they should.

Even though you can create identically named files in different directories, I'd strongly recommend against it. At some point in the art process, you will be revisiting the files again, and it becomes a very confusing, error-prone process if you are calling up similarly named files. Another reason might occur months down the development road. An application might end up being written that helps sort and export your art data very efficiently. These applications can save you days of work, but generally are more reliable if you use unique identifiers. There are always workarounds to the problems you will face, but it is good practice to err on the side of flexibility.

So bite the bullet and go back and rename the files with a more logical naming scheme, using the directory structure that you've created as a basis for the file names. After creating the first file called 01METER_DECIDUOUS_MAPLE_TREE01, you realize that some people might balk at a 30-letter file name. Some data truncating might be needed to make the file more manageable, so you cut the file down to 1M_D_M_T01. Knowing you are on the right track, you blaze through the rest of the files and update them to your new scheme.

Once again, things are right in the world of GUMBO. The art progresses along swimmingly for another week before you realize that you need to define your trees further with light and dark versions. Your first forays into the new detail level start to show some problems with your naming scheme. The file 1M_D_M_T01_D looks O.K. until someone points out that the first "D" stands for "deciduous," and the second "D" stands for "dark." So you rename it 1M_D_M_T01_DK. It all makes perfect sense until you hand the file off to the new guy who started last week and his

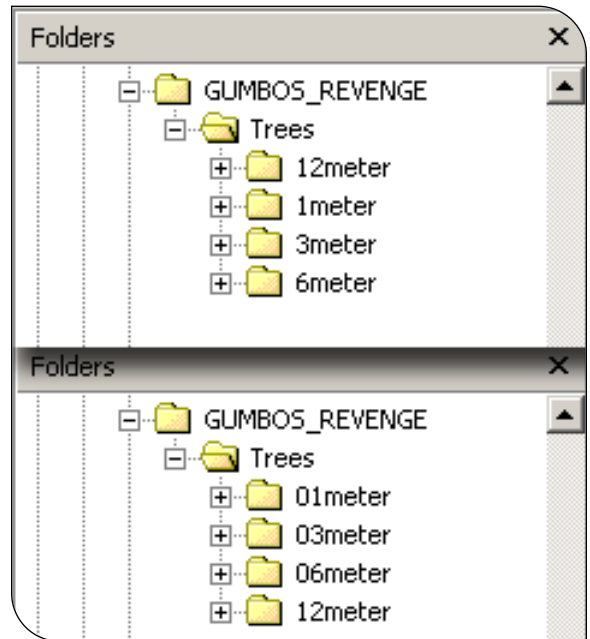


FIGURE 2. An example of how the addition of a zero can cause directories to sort correctly.

eyes glaze over as he looks at your "logical" file name.

Now, take that one example, and multiply it by a hundred, or a thousand. Do other assets in the game follow the same logical structure as the trees? How about houses? Character files? Is it going to take you an hour to explain your naming scheme to someone? If so, can you make it more efficient? Have you checked with the programmers to see if your scheme works with the way they are writing the code?

It quickly becomes clear that you will need to think this process through carefully and keep a close watch on it as the project moves forward. When done well, this is one of the less visible infrastructure components that allows the development process to proceed at optimal speed. When done poorly, it can quickly drag the art asset process into a quagmire of problems.

"I'll Take Technology for 600, Alex"

As with many things in this "new" business of game development, the problems we face have often been dealt with before, only under different circumstances. Take, for example, organizing a library full of books. It's essentially the same problem,

just applied to books instead of computer files. Most of us have heard of the Dewey decimal system used by libraries to organize their materials. Just so you know (and I had to look this one up too), it's based on the division of all knowledge into ten groups, with each group assigned 100 numbers. The 10 main groups are: 000 to 099, general works; 100 to 199, philosophy and psychology; 200 to 299, religion; 300 to 399, social sciences; 400 to 499, language; 500 to 599, natural sciences and mathematics; 600 to 699, technology; 700 to 799, the arts; 800 to 899, literature and rhetoric; and 900 to 999, history, biography, and geography. These ten main groups are in turn subdivided again and again to provide more specific subject groups. Within each main group the principal subseries are divided by ten as well.

O.K., now that we've had our history lesson for the day, let's get back to the problem at hand. Why couldn't we just apply the same technique to the issues of file names and be done with it? From the

computer's standpoint, a unique number or letter-number combination is no different from a long, descriptive word. All that is generally required is that it be unique from any other file. Are there distinct advantages or disadvantages to using a numbered system? The answer is, it depends.

A numbered system can work well, and is quite flexible, but it isn't without some costs. One of the main disadvantages is that you have no way of visually sorting the files in a directory without either seeing them (via opening them up or thumbnails) or using a lookup table.

When given the chance, most of us are likely to label a picture of a flower as something descriptive, such as FLOWER1.JPG as opposed to G32AL321.JPG. This is just common sense, since you want to be able to look at a directory of files and be able to pick out your flower file easily. If the file is named something that has no recognizable reference to your original file, it becomes much more difficult to locate and identify.

Once you have waded through two or three directories that are nothing but numbered files, it becomes impossible to remember what the files represent.

So why would you choose this method at all? One reason might be if you have files that are constantly being shuffled, reassigned, or changed. If this occurs, it's often necessary to rename the files continually in order to maintain the integrity of the organization structure you have built.

For example, let's say you have a file called TOWN_INN_BUCKET.PSD. Under a directory-driven structure, this represents a picture of a bucket, found in the "Inn" subdirectory, which is located in the "Town" directory. Later on in the development cycle, you realize that this bucket is much more applicable for use in the barn, which is located in the countryside section of your game. If you move the file to the new directory, the filename doesn't match the existing structure. Renaming isn't difficult, but it takes a small bit of time.

Let's say that your bucket item needs to be more of a global type. So you create a "Global" directory, and break it down into logical subdirectories. Again, you move the bucket file and because its name is derived from the directory it resides in, it needs to be renamed. While this is a simplified case, changes like this can occur on a daily basis as a project continues.

If, on the other hand, the file was originally just given a unique number or letter-number identifier, it could be shifted around easily without you having to worry about renaming it. This can be very advantageous if there are a lot of files prone to this type of evolution.

The main disadvantage is that a lookup table of some sort will have to be maintained and readily available as a cross-reference. Generally, this requires several spreadsheets that must be constantly maintained. In addition, error-checking is sometimes difficult to do since it is easy to transpose characters that have no specific logic to their structure. If you misspell FLOWER1.JPG, it's easier to see and correct than if you mistype G32AL321.JPG.

Make a Key or Two

One of the first things I do when beginning a naming scheme is to start some "living" documents. These are ones that are constantly updated as the project progresses. They should be in a location that is readily accessible to the other team members, which can be as simple as a shared directory over a LAN, or it can be more tightly controlled by using some sort of version control such as Visual SourceSafe.

A good first document to create is a key for others to decipher your naming scheme. This will be a requirement if you use a numbered scheme. If it is a directory-based naming scheme, then it isn't required, but it will certainly make life easier if it exists. Since most descriptive words you use in your file are going to get truncated, you should have a document that everyone goes to when they are looking for the abbreviation of a word. Even if you think the word is so simple that no one could possibly condense it incorrectly, you will be amazed. Let's take the word GOLD for example. Is it GLD, GL, or GD? If you leave it up to interpretation, the chance for error is high.

What Should I Use?

Generally, I try to stick with the basics. As a computer artist, you may not be well versed in programs that don't directly deal with creating art assets. While this is fine for production-level positions, you will find that your software knowledge base needs to expand as you take on the responsibility of managing assets.

Spreadsheets are a great type of program to get familiar with. They are like the Swiss army knife of applications. I typically use Excel, mainly because it is compatible with just about everyone, and because I am developing on the PC environment, it is one of the base tools I have at my disposal. It's powerful, flexible, and easy to learn. It's also available to all of the artists on the team as part of their basic system software, so I don't have to concern myself with additional software purchases. Choose a program that everyone has access to; otherwise they won't be able to review, edit, or update your files without help.

Once you get into the management side of the assets, you will soon find yourself in the position of needing to supply data to the person(s) responsible for tracking the progression of your project. With a properly organized spreadsheet, you can find and present this information easily. Excel also allows linking to other spreadsheets. One way to take advantage of this capability is to create linked summary sheets. If properly set up and linked, summary spreadsheets can be made to update automatically from other spreadsheets. This is a great way to create files that are always current (assuming the linked files are maintained) and usually provide the executive summary information needed without all of the detail.

Safety Issues

As my example at the beginning of this column showed, poorly organized files can lead to some serious flaws in the backup process. Part of any good development structure includes the consistent backup of both the final "CD image" files and the production art files. Production files are those which are used to


create the final game assets. In the case of Photoshop, a production file can often be 20 or 30MB in its nondestructive, uncollapsed form while the final asset is under 1MB. Also, most artists keep several iterations of files on hand. Animations are another good example. They are exported from the 3D tool of choice, creating a game-specific final asset.

It's easy to see that a lot of files need to be considered when thinking of the safe backup of the artist's development environment. If it follows a logical, well-thought-out structure, this can be a very easy thing to manage and maintain. If not, files can go for months or years without being backed up, and only when a hard drive fails or data turns up missing do you realize the files are missing from what you perhaps thought was a safe system.

So what backup method should you use? As usual, it depends on your project. Most big development teams will have an automated LAN backup of some type. Data is backed up every day or two, and as long as you have your critical files on the network, you are safe. CD backup burns that include your iterations and legacy files are a good idea. Like most artists, I usually discover a particular file has something useful in it the day after I delete it. If you intend to burn CD backups, think about how best to organize your directories. It's a lot easier to copy and burn one "backup" directory tree than it is to spend an hour gleaning the files out of your working structure.

Plan Ahead, Stay Ahead

As the person responsible for managing and/or maintaining art assets, you will find that the organization of your data is an ongoing process. If you set up a good system, it doesn't have to be a full-time job to update and maintain.

While file organization isn't the most glamorous part of game development, it is important, and can have a major impact on how well your project runs. Remember to keep it flexible, and keep it simple. 



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



Achieving Real-Time Realistic Reflectance

JAN KAUTZ | Jan is a Ph.D. student at the Max-Planck-Institute for Computer Science in Saarbrücken, Germany. His main research area is interactive realistic lighting and shading using graphics hardware. He can be reached at kautz@mpi-sb.mpg.de. **CHRIS WYNN** | Chris is an OpenGL software engineer working at Nvidia Corp.'s technical developer relations group. You can ask him anything (BRDF and otherwise) at cwynn@nvidia.com. **JONATHAN BLOW** | Jonathan prefers the AK-47 and the Colt M4A1. He will use the MP5, though, if that's what it comes down to. He reads e-mail sent to jon@bolt-action.com. **CHRIS BLASBAND** | Mr. Blasband has more than 17 years of experience in applying BRDF phenomenology to military and commercial applications. As president of Soreal Technologies, he is responsible for deploying Surface Optics Corporation's technology into commercial off-the-shelf products. He can be contacted at cbb@soreal-tech.com. **ANIS AHMAD** | Anis is an undergraduate student at the University of Waterloo, majoring in computer science. You can contact him at a3ahmad@student.math.uwaterloo.ca. **MICHAEL MCCOOL** | Michael, who can be reached at mmcool@cgl.uwaterloo.ca, is an associate professor with the Computer Graphics Lab at the University of Waterloo, Canada. His research areas include real-time hardware-accelerated shading and illumination.



One of the primary goals of game development is to successfully convince players that they are in a different world — this is particularly important for 3D and first-person games where visual immersion may mean the difference between life and death, victory and defeat, or first place and second. In order to achieve this visual immersion, the lighting must be as sufficiently convincing as all the other elements in the environment. That is, the quality of the geometric models and the lighting simulation should be good enough to trick a player into suspending reality and believing that he or she is in the fantasy world that months of development time were devoted to creating.

The increased fill rates and optional transform and lighting engines that highlight the evolution of 3D accelerators have paved the way for increasingly complex geometric detail. What once was unimaginable is now commonplace, as games that utilize the potential of these accelerators make their way through the production pipeline. The good news is that this trend of increased polygon throughput is likely to continue in the future. Unfortunately, however, the quality of lighting has not improved at the same rate, and many games still lack the visual authenticity necessary to provoke a truly compelling gaming experience. Artifacts resulting from poor vertex lighting continue to abound in games, and while the geometry improvements have helped the lighting cause, our games do not yet exhibit many of the lighting phenomena observable in the real world.

So what's necessary to achieve realistic lighting? To answer this question, two areas must ultimately be addressed: illumination and reflectance. Illumination determines the distribution of light striking the surfaces in the scene, and includes the effects of shadows and indirect lighting. Reflectance determines how surfaces redistribute the light that hits them. For realistic lighting, both the illumination and reflectance need to be simulated.

Within the game development community, several current approaches address the illumination problem. Point lights (with optional fog, distance, or shadow attenuation) are often used to determine the amount of light that arrives at a surface. Directional light sources and light maps effectively serve this purpose as well. Unfortunately, sophisticated models of reflectance have not really made an appearance in games. In terms of reflectance, most games to date use the Phong reflectance model or rely on strict intensity modulation to determine how surfaces reflect the light that strikes them. While this is not a bad thing, Phong reflectance and intensity modulation are limited in the types of lighting phenomena they are capable of simulating. Consequently, they are unable to reproduce the appearance that we observe of many real-world materials.

This two-part series of articles focuses on the reflectance aspect of lighting. We will discuss a technique that implements more general reflectance models for a wide variety of surface materials, for

example velvet, copper, and others. This is called separable decomposition and is an effective and efficient way to incorporate physically accurate reflection models and ultimately increase the level of realism in a game. The technique can be used in conjunction with point light sources, directional light sources, light maps, shadows, and fog, since each of these influences only the illumination component of lighting and does not affect the reflectance model. Moreover, the technique evaluates reflectance on a per-pixel and not a per-vertex basis, and it can support dynamic lights with no additional overhead. Finally, while the approach is general and allows for arbitrary reflection models, it is still capable of simulating Phong if desired. Figures 1 through 4 demonstrate examples done at interactive rates with our technique.

This month, we will cover the mathematics behind reflectance as well as how to acquire reflectance information from real-world materials. Next month, we'll detail our separable decomposition technique and how to implement it using multi-texturing.

Of course, a real application wouldn't use just, say, brushed metal, but rather brushed metal that's scuffed, rusty, dirty, has glowing green alien mucus smeared on it, and has holes punched in it where acid has eaten through.

Scope of This Article

As we said, the process of lighting involves both illumination and reflection. In this article we will concentrate on point-source direct lighting, where illumination comes from a point positional or directional source and is reflected by a surface directly into the eye. Even applications that use sophisticated precomputed global illumination should not neglect direct lighting from the brightest light sources, since such lighting conveys the bulk of the information about the shape and surface qualities of objects in the scene.

Many if not most current-generation rendering engines add interest to surfaces by using color, bump, environment, and/or specular maps. While these techniques are very effective, they are still limited by the underlying reflectance models they modulate.

For simplicity, our examples will focus on materials with homogeneous surface properties; that is, each triangle in a model can represent only one type of material (such as metal or wood). Of course, a real application wouldn't use just, say, brushed metal, but rather brushed metal that's scuffed, rusty, dirty, has glowing green alien mucus smeared on it, and has holes punched in it where acid has eaten through. What we're going to present is not merely another technique to put in your bag of tricks, but a technique that will greatly enhance other effects when combined with them.

Reflectance Models

In computer graphics, when we talk about materials or material properties, what we are really talking about is the reflectance properties of a surface that define how light arriving at the surface is scattered. A reflectance model can be thought of as a material



LEFT TO RIGHT: FIGURE 1. Metallic anisotropic material. FIGURE 2. Measured peacock feather: note color shifts at certain viewing angles. FIGURE 3. Analytical BRDF for gold with an additional diffuse texture. FIGURE 4. Measured wood with an additional diffuse texture.

description that modulates the intensity of the light that arrives at the surface. As parameters, it takes the angles by which incoming light arrives at a surface point and the angles by which it must leave that point to enter the eye; as outputs, it gives us the manipulations that should be done to the light to simulate the reflective properties of a material. Currently, the Phong reflectance model is used almost universally for “glossy” surfaces in real-time rendering. Phong reflectance is easy to use because it is directly supported by existing graphics APIs such as DirectX and OpenGL. Moreover, in 3D accelerators that include hardware transform and lighting support, the computation of Phong reflectance can be performed very rapidly and does not burden the general CPU.

Unfortunately, the Phong reflectance model is not physically realistic, and many real surfaces have reflectances that look nothing like it. Furthermore, the current implementations of the Phong lighting model in OpenGL and DirectX both have some severe problems because they evaluate the reflectance model only at the vertices of the polygon instead of at every pixel sample rendered. This method of evaluating a reflectance model at the vertices of a primitive and then interpolating the resulting colors across the primitive is the commonly known Gouraud shading technique. Gouraud shading is fast and requires far fewer computations than evaluating the reflectance model at each pixel, but unfortunately it can be problematic — particularly so for shiny surfaces in which the reflectance varies rapidly with view and illumination direction, and contains specular highlights that do not fall on a vertex in the model. For example, if the object being lit is modeled using a small number of polygons, each covering many pixels, a rapidly varying reflectance function can be undersampled, resulting in aliasing of the reflectance model. This can result in splotchy highlights that may flash on and off as the object moves, which does not convey the desired impression of a shiny surface. Hence it is desirable to have a technique that performs the lighting computations on a per-pixel basis rather than on a per-vertex basis.

Finally, the Phong reflectance model is isotropic: the amount of reflectance depends only on the elevation angles; that is, the angles that the view and light directions make with the normal. General reflectance models are anisotropic: they also depend on the azimuth angles that the view and light directions make with some preferred tangent embedded in the surface and perpendicular to the normal. In other words, anisotropic reflectance models have an orientation about the normal, not just an elevation angle. Examples of anisotropic materials include brushed metal and most woven fabrics. Some materials, such as satin, are extremely anisotropic. Other materials have a highly anisotropic

reflectance when viewed from a distance, such as hair, fur, and mown grass. Even with a static light source and viewer, a surface with an anisotropic reflectance will reflect differently when rotated around its surface normal. See Figure 1 for an example of an anisotropic brushed material.

The BRDF

The Phong reflectance model became popular because it was easy to compute and produced fairly attractive results. In order to develop more sophisticated reflectance models that do not suffer from Phong’s problems, we need to define mathematically what a “reflectance model” is. The abstraction used by computer graphics researchers is the bidirectional reflectance distribution function, or BRDF.

When light reflects off a perfectly smooth specular surface, the angle of reflectance is equal to the angle of incidence. However, perfectly smooth surfaces don’t exist in the real world. Rather, surfaces tend to be quite complex at a microscopic level. For example, a wood surface, no matter how polished and smooth it may feel, contains millions of tiny fibers packed together. When light reflects from surfaces like these it scatters in many directions. Depending on the type of surface, light will be more prone to scatter in certain directions. Directions in which a large fraction of the incoming energy scatters are usually seen as highlights. A BRDF models the probability that a photon from a certain incoming direction will scatter in a certain outgoing direction, or will be absorbed by the surface. An example of such a distribution (for one light direction) can be seen in Figure 5.

The BRDF can be used to compute, for a particular view direction, the outgoing radiance (which corresponds to what the eye actually perceives) as a weighted integral over all incoming radiance at a particular surface point:

$$L(\theta_o, \phi_o) = \iint_{\Omega} f(\theta_o, \phi_o, \theta_i, \phi_i) L(\theta_i, \phi_i) \cos(\theta_i) d\sigma(\theta_i, \phi_i)$$

Here f is the BRDF and L represents the radiance traveling in the direction given by the spherical coordinates θ and ϕ relative to the surface point. The elevation θ is the angle between a direction and the surface normal, the azimuth ϕ is the angle of a direction around the normal. The spherical coordinate system is shown in Figure 6. The subscripts o and i represent the outgoing (view) direction and incoming (light source) direction respectively. The notation $d\sigma$ (the solid angle measure) just indicates that all directions on the hemisphere Ω should be weighted equally.

If only point sources are considered, then light can only arrive from specific directions, and this reduces to the following reflectance equation for K point light sources:

$$L(\theta_o, \phi_o) = \sum_{k=1}^K f(\theta_o, \phi_o, \theta_{i,k}, \phi_{i,k}) L_k(\theta_{i,k}, \phi_{i,k}) \cos(\theta_{i,k}) \quad \text{Eq. 1}$$

In the following, we will use normalized vectors $\hat{\omega}_i$ and $\hat{\omega}_o$ to represent the incoming and outgoing directions, in addition to θ and ϕ for elevation and azimuth angles.

In addition to the incoming and outgoing ray directions relative to the surface, BRDFs often also depend on the position on the surface and the wavelength of the incoming light. For the purposes of this article, we will deal only with the incoming and outgoing directions, which reduces the total number of parameters to the four angles. Rather than use the wavelength of light as a parameter, we normally will evaluate red, green, and blue components of the BRDF separately, and combine these values into a final color.

Reflectance Equation and Phong Model

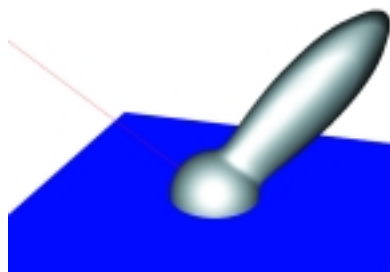
Equation 1 can be understood more easily if we examine how a raytracer usually evaluates this equation and how it relates to the well-known Phong reflectance model. A raytracer traces rays from the eye point through every pixel of the view plane. Whenever a ray hits a surface at point \bar{p} , the normalized viewing vector $\hat{\omega}_o$ (pointing from \bar{p} to the eye) and the normalized light vector $\hat{\omega}_i$ (pointing from \bar{p} to the light source) are computed. The incoming radiance $L_k(\hat{\omega}_i)$ from the k th light source is computed by dividing the intensity of the light source by the squared distance from \bar{p} to the light source. The incoming radiance $L_k(\hat{\omega}_i)$ is then multiplied with the dot product between the surface normal \hat{n} and the light vector $\hat{\omega}_i$, which corresponds to $\cos(\theta_i)$. The result is then multiplied with the BRDF $f(\hat{\omega}_o, \hat{\omega}_i)$, which is the radiance value perceived by the eye. Rewriting the reflectance equation (Equation 1) using vectors and only one light source gives this:

$$L(\hat{\omega}_o) = f(\hat{\omega}_o, \hat{\omega}_i) L_k(\hat{\omega}_i) (\hat{n} \cdot \hat{\omega}_i) \quad \text{Eq. 2}$$

How does this correspond to the Phong model used by OpenGL, for example? The Phong model consists of diffuse and specular components and can be written like this (simplified):

$$L(\hat{\omega}_o) = \left(k_d (\hat{n} \cdot \hat{\omega}_i) + k_s (\hat{n} \cdot \hat{h})^N \right) L_k(\hat{\omega}_i),$$

FIGURE 5. Reflectance lobe for a sample reflectance function.



where the halfway vector $\hat{h} = \frac{\hat{\omega}_i + \hat{\omega}_o}{|\hat{\omega}_i + \hat{\omega}_o|}$. The intensity of the diffuse

part can be changed with k_d , the intensity of the specular highlight can be influenced with k_s , and the size of the highlight can be adjusted using the parameter N . We can rewrite this equation to show the relation to the reflectance equation (see Equation 2):

$$L(\hat{\omega}_o) = f_{\text{phong}}(\hat{\omega}_o, \hat{\omega}_i) L_k(\hat{\omega}_i) (\hat{n} \cdot \hat{\omega}_i)$$

$$f_{\text{phong}}(\hat{\omega}_o, \hat{\omega}_i) = k_d + \frac{k_s (\hat{n} \cdot \hat{h})^N}{(\hat{n} \cdot \hat{\omega}_i)}$$

So basically the Phong model is nothing but a specific BRDF, which is fairly simple to evaluate. It should be noted that the parameter k_d is very often stored in a “diffuse texture map,” and the parameter k_s is often stored in a so-called “gloss map.”

Better Reflectance Models

Computer graphics researchers have developed many sophisticated reflectance models that can better represent real surfaces. Most of these models are derived directly from the underlying physics of optics and mathematical models of light-surface interactions. To discuss even one of them in any detail would require an article of its own. We will, however, mention a few of the more interesting models available, references for which are provided at the end of this article. Fortunately, you don’t have to understand the physics to use these models, and the separable decomposition technique we will present in next month’s article uses sampled data and can handle all these reflectance models at real-time rates.

The Torrance-Sparrow model is a variant of a physically based reflection model for isotropic surfaces. It assumes that each surface consists of tiny, randomly oriented facets, each of which is a perfect reflector (see Heidrich and Seidel under References). The He model is based on physical wave optics. This isotropic model is extremely powerful, as it can describe many surfaces from roughened rubber to polished gold. It is also complicated, difficult to implement robustly, and computationally expensive. Ward’s model uses anisotropic Gaussian lobes. Many other reflectance models can be represented to reasonable accuracy by summing multiple Ward lobes. The Poulin-Fournier model assumes microscopic cylinders cover the surface parallel to one another to model anisotropic materials.

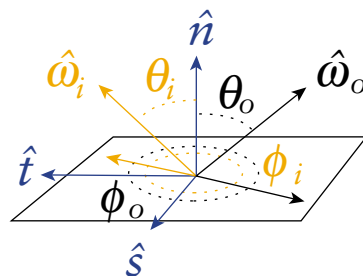


FIGURE 6. Visualization of the angles and vectors used: \hat{n} = normal, \hat{t} = tangent, \hat{s} = secondary.

Unfortunately, the above models aren't directly suitable for hardware-accelerated real-time applications, as they don't address the problem of implementation. However, measured data is available for several materials (see Columbia-Utrecht Reflectance and Texture Database, and the Cornell University Program of Computer Graphics' Measurement Data under References) and can be used instead of analytical models. This reflectance data can be used in the approach we'll discuss in next month's article, and we'll give some examples. A BRDF can also be generated from computer models of micro-geometry and bump maps (see Westin and others under References). The second approach can be used to render objects with bump maps properly at a distance.

Acquiring BRDF Data

Currently, commercial BRDF data is only sparsely available and focused mainly on measurements of materials of military significance. Still, a great deal of research has been performed on acquiring BRDF data and there are companies today that are building databases, selling instruments, and performing BRDF measurements for commercial applications.

There are three principal means of arriving at BRDF measurements: two are currently in use, and the third is in the experimental stages of development. They are a goniometer, the newer imaging bidirectional reflectometer, and an image-based BRDF measurement derived from photographs of an object.

The principle behind all of these instruments is that a sample of material is illuminated by a known light source and the amount of scattered energy is measured as a function of the elevation and azimuth angles of the light source, elevation and azimuth angles of the observer/detector, and the wavelength of the incident light source. Therefore, the data produced by these instruments is truly five-dimensional.

While it sounds simple in theory, the actual practice of building instruments to measure a calibrated BRDF is quite difficult. It requires a detailed knowledge of electronics, optics, and scattering theory.

Sample preparation is critical for many materials, and is quite often overlooked by those making BRDF measurements. For example, dirt placed in a pile will have quite different scattering properties from dirt that is thinly spread in the instrument cup. Material such as velvet can have very different scattering effects if placed in a tightly stretched position versus loose or crumpled. The moisture content of the material is also important. A wet material will scatter light very differently from when it is dry.

Measuring the BRDF is as much an art as a science. The sample must be properly prepared and situated in or under the BRDF measuring device. It is also critical that the person performing the measurements document the condition of the material when measured.

BRDF Measurement Devices

Goniometers. The most well-known device for measuring the BRDF is a goniometer. A goniometer performs an accurate BRDF measurement by performing a systematic mapping of the



FIGURE 7 (above). SOC-250 in situ bidirectional reflectometer. FIGURE 8 (right). SOC-200 laboratory bidirectional reflectometer.



light scattered in the hemisphere. It does so utilizing a movable light source and sensor, making a measurement for each incident and exit angle of light on the surface. This gives a true five-dimensional (four angles and wavelength) mapping of light scattered by a surface. Figure 7 shows a smaller BRDF system in use at the U.S. Army's Tank-automotive and Armaments Command (TACOM) for laboratory and field measurement. Figure 8 shows a second, and much larger, fully automated system used in the laboratory at the U.S. Navy's Carderock Division Naval Service Warfare Center (CDNSWC) to measure larger samples.

It is important in the design and use of these instruments that one has knowledge of the type of light being used to illuminate the material. Illumination sources for the measurements include quartz halogen and blackbody sources (for examining BRDF characteristics in the infrared) and up to five laser sources to cover specific wavelengths in the visible (for example red, green, blue). A number of detectors are employed to provide continuous measured data from the visible to the long-wave infrared.

The information measured by these instruments is a five-dimensional matrix based on incident and exit angles of light from the surface and the wavelength of the illumination source. Figure 9 shows a three-dimensional plot of the BRDF for all exit angles and just one incident direction $(\theta_i, \phi_i) = (50^\circ, 0^\circ)$ of an Army green paint at a wavelength of $0.5\mu\text{m}$ (green light) as measured by one of these instruments. As you can see, there are quite a few measurements that must be made in order to accurately characterize the BRDF of a material.

Imaging bidirectional reflectometers. A second type of instrument, the imaging bidirectional reflectometer, has been developed to complement goniometers, and provides a more cost-effective and convenient means of measuring BRDFs in the field with only a slight loss in accuracy. These new handheld BRDF imaging devices allow developers to measure BRDFs in real time at their own facilities and without the need for a laboratory.

One such device is capable of simultaneously measuring both BRDF and hemispherical directional reflectance (HDR), a measure of the total reflectance as a function of the incident illumination angle of a surface. Using an imaging technique that takes into account the elevation and azimuth angles of the light source, and an imaging array that essentially requires no moving parts as in the goniometer, this instrument fully maps the scattering from a half-hemisphere above the surface with more than 30,000 angularly resolved points and update rates to 60 measurements per second. The user just places the instrument against the surface to be measured and presses a button. The light source elevation angle is variable from $\theta_i = 0$ to 85 degrees, while scattering is measured to nearly 90 degrees off normal.

Each pixel in the "angular image" (hemisphere) formed on the array corresponds to a small range of scattering angles relative to the illuminated spot on the measured surface, as shown in Figure

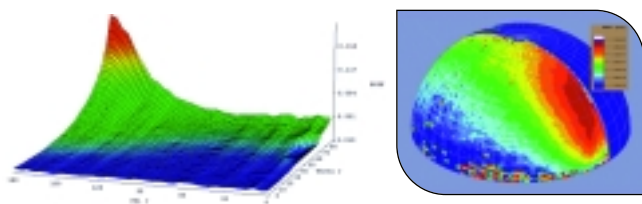


FIGURE 9 (left). BRDF of Army green 383 at 0.5µm and 50° elevation angle. FIGURE 10 (right). Angle image formed by the handheld HDR instrument.

10. If the detector response is known for each pixel, the images may be interpreted as a quantitative measurement of the distribution of the reflected light. Alternatively, dividing by the incident intensity of the illuminating beam, the image can be interpreted as revealing the distribution of reflectance from the sample, which is exactly what is needed.

Image-based BRDF measurement. A third method, image-based BRDF measurement, is a means for measuring the BRDF at visible wavelengths. Using photographs of an object and known camera position, lighting, and geometry effects, the BRDF of the object's surface is determined directly from the photographs. In essence, without specialized apparatus, game developers can use the camera's positioning and knowledge of the conditions surrounding the taking of the picture to derive some of the parameters describing the BRDF.

This methodology is currently under investigation (see Marschner under References) and could be very cost-effective, accepting some further loss in accuracy. It holds promise, but as yet no commercial systems exist, and it might require further validation.

BRDF Measurements for Game Developers

The imaging bidirectional reflectometer is the instrument best suited for game developers. It is extremely user-friendly and requires little effort to operate. Simply place the aperture on top of the material to be measured, adjust the illumination angle, and push a button. The result is a matrix of 30,000 BRDF data points for that sample. The data is stored in a documented format on the computer that comes with the instrument. It is then very easy to transfer the data to any computer.

For those who do not wish to make their own measurements, there are companies developing commercial BRDF databases composed of many materials relevant for today's games. Developers can also hire one of these companies to develop custom BRDF databases for their specific applications. Some public BRDF measurements can also be found online, such as Cornell University's and the Columbia-Utrecht Reflectance and Texture Database mentioned previously.

Coming Next Month

In this article, we have presented the necessary background for the separable decomposition technique, which we will explain in detail in part two next month. This technique allows you to render realistic surfaces with arbitrary reflectance models. The reflectance models used can be either analytical or measured with one of the presented devices. 🎨

REFERENCES

- Kautz, J., and M. McCool, "Interactive Rendering with Arbitrary BRDFs Using Separable Approximations." *10th Eurographics Rendering Workshop, 1999*. pp. 281–292.
- Heidrich, W., and H. P. Seidel. "Realistic, Hardware-Accelerated Shading and Lighting." *Proceedings of SIGGRAPH 1999*. pp. 171–178.
- He, X., and others. "A Comprehensive Physical Model for Light Reflection." *Proceedings of SIGGRAPH 1991*. pp. 175–186.
- Ward, G. "Measuring and Modeling Anisotropic Reflection." *Proceedings of SIGGRAPH 1992*. pp. 265–272.
- Poulin, P., and A. Fournier. "A Model for Anisotropic Reflection." *Proceedings of SIGGRAPH 1990*. pp. 273–282.
- Columbia-Utrecht Reflectance and Texture Database
www.cs.columbia.edu/CAVE/curet
- Cornell University Program of Computer Graphics
www.graphics.cornell.edu/online/measurements
- Westin, S., J. Arvo, and K. Torrance. "Predicting Reflectance Functions from Complex Surfaces." *Proceedings of SIGGRAPH 1992*. pp. 255–264.
- Marschner, Stephen Robert. "Inverse Rendering for Computer Graphics". Ph.D. diss., Cornell University, 1998.

Calling the Shots

Decision Making for Games



Here's our game, DEPRAVED INTERN'S REVENGE, ready for publication.

Great! Now just add 10 more levels and multiplayer.

Let's get a license – here's a list of what is available. Just pick something.



You've heard it before: the story about some important decision that went terribly wrong. If you're lucky, it just leads to angst and hours of extra work. If you aren't so lucky, it leads to your project being killed.

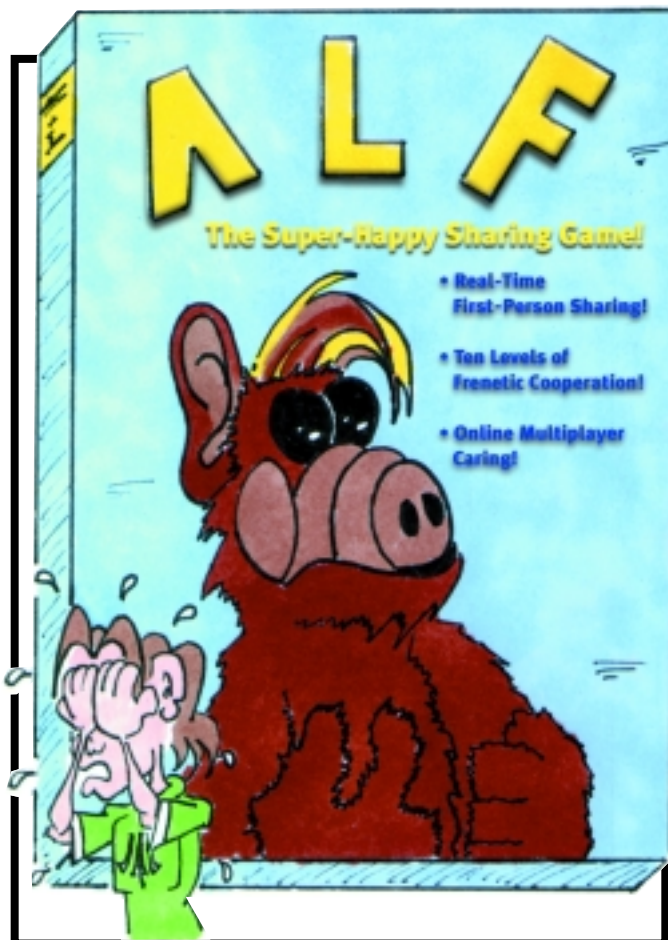
Sometime after the crisis is over, a meeting is held in which everyone reflects on what went wrong. Usually this amounts to passing the buck and looking to avoid blame. The meeting ends with comments about "learning from our mistakes" and "we'll know better next time." Rarely does such a meeting get to the real root of the problem, though.

What needs to happen in a case like this is for the group to take a long, hard look at how the problem-causing decision was made. You need to talk about the process of decision making and figure out how to change the processes that led to that decision. For this article, I'll divide decision making processes into a cou-

ple of categories that are particularly relevant to game companies. The first division will be the number of people who make the decision: either one person makes the decision, or a group makes the decision. The second division is the basis for making the decision. The two categories of this second division are "political reasons" and "merit-based reasons." Combining these two categories, we can create a two-by-two matrix, as seen in Figure 1.

Sometimes a process might pop up that doesn't fit these divisions very well (you might just flip a coin, for example), and other times there might be some blending of these processes, but for our purposes these categories will suffice. One exception is worth noting, however: the case where no decision making process exists. People just put whatever they want into the game, no one decides what is good or bad, and when the schedule says the game is done, they ship it.

CHIP BURWELL | Chip is the owner of Gratuitous Games Inc., a small development studio currently working on a race game for PS2, Xbox, and Gamecube. Chip started in the industry as a sound guy but after a stint working on Project Reality at SGI, he moved into game programming. Recently, Chip has been studying drawing to develop his understanding of art. Learn more about Gratuitous Games at www.gratuitousgames.com, or contact Chip at chipb@gratuitousgames.com.



ILLUSTRATIONS BY R.D.T. BYRD

Keep in mind that these divisions are hardly the only categories you can use to identify decision making processes.

There is extensive literature on this topic (see For More Information for some good references), and a number of different theories. To discuss all the theories of decision making would be beyond the scope of this article, nor is this article meant to be a synopsis of these theories. It should be a starting point for conversations among game developers about how they work together.

Single-Person Decision Making

In cases where the “big cheese” decides, there is one person who has been designated the decision maker. Everyone knows who this is, and whatever this person decides, that’s the way it is. This method has a couple of distinct advantages. First, it’s relatively easy to get a decision made, and the group knows whom to look to for its decisions. Because just one person is making all the decisions, there is a uniformity to the end results. Also, decisions can be made very quickly, since no meetings need to be called.

But the biggest strength of the big cheese process is also its biggest weakness. That is, the judgment of the big cheese. If the right person is the big cheese, then this method can go well. But if the wrong person is the big cheese, it can be a disaster.

On one game design I worked on, we had a producer who was regarded as one of the best in the industry. The producer’s idea involved looking down on gameplay in a 3D environment. When we first got a demo of the gameplay up and running, virtually everyone on the team found it wasn’t much fun. The main issue was that viewing the game from a top-down angle, it was extremely hard to tell what height your character was at. But the producer was unable to let go of his original idea, and even encouraged us to increase the height of the game world, making things

Have you guys had a chance to work on my “4-player, second-person perspective, first-person shooter” idea yet?



worse. In the end, much to the team’s relief, the project was canceled. The lesson here is that, under the big cheese method, you live and die by your cheese.

There is another aspect of the big cheese method that is often overlooked, which is the impact this method has on the people who work under the big cheese. First, because decisions are always being made for them, the people working under the big cheese don’t get experience making decisions. That’s too bad, because there is something to be said for learning from your own mistakes. If your company is trying to develop the talents of its staff, this will be a big shortcoming.

Second, when all decisions are made for them, creative people will find the

environment un motivating. At a minimum, this leads to people not having a personal involvement with the project. Worse, it can lead to creative people leaving and finding jobs elsewhere. But if you have someone who is extremely talented and has excellent judgment, it would be a shame not to use them as your decision maker.

Despite these shortcomings, there are lots of times when having a single decision maker is a real benefit. In a crisis situation, having a strong leader who makes clear decisions and gives straightforward instructions can be invaluable.

Group Decision Making

Before discussing group decision making, I should warn you that I used to believe that group decision making was always a bad idea. My first job in the game business was as a sound and music guy. Because there was no “music director,” the decision about what music would stay in the game and what would get cut was based on comments from the project lead, the CEO, the art director, the marketing department, and various others. I found that any song that might bother anyone was likely to get cut. The songs that were chosen to stay in the game were often the least inspired and most banal.

After a while, I stopped writing music and got more involved with game programming. Working in small groups with other programmers, I found that we could make good decisions together. Typically the group suggested several different ideas, debated their relative merits, and finally chose the best idea from the group.

Once I experienced this, I began wondering why some groups make good decisions and others don’t. At first I thought it was because programming decisions tend to be objective, while others, such as music decisions, are subjective. But some programming decisions are subjective, and I had seen programmer groups handle these decisions well, too.

DECISION MAKING STYLE MATRIX	
Single decision maker using politically based decision processes	Group decision making using politically based decision processes
Single decision maker using merit-based decision processes	Group decision making using merit-based decision processes

FIGURE 1. A matrix of decision making processes relevant to game development.

As time went on I became more involved in managing and leading a team. More decisions were being placed on my shoulders, and I became increasingly concerned with how our company was making decisions. I began reading about how organizations function, and trying to relate what I read to my experience. The key element that kept showing up in both my reading and my experience was the nature of the group itself. Time and again this seemed to be the determining factor in how successful a group was at making good decisions.

Is It a Group or Is It a Team?

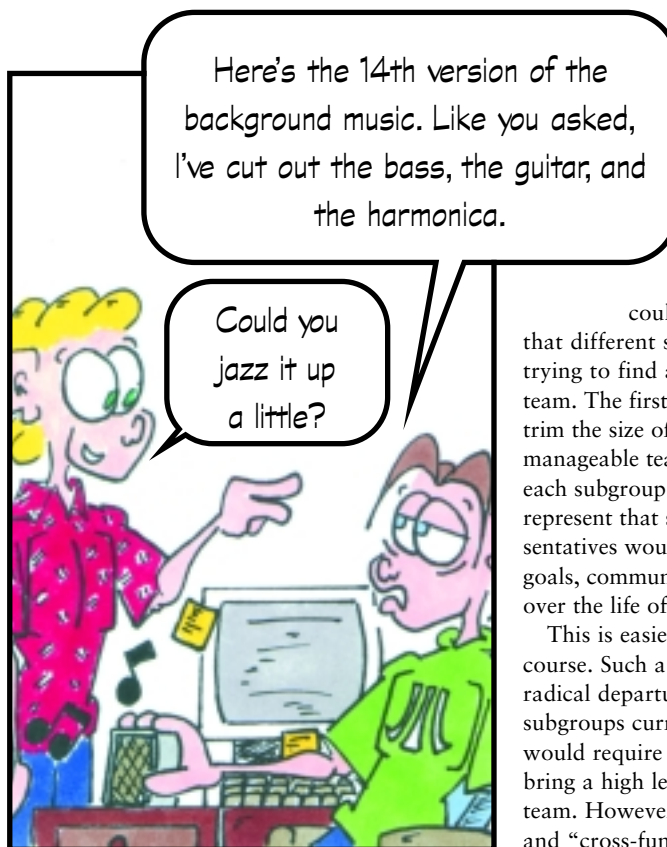
Now is a good time to introduce the distinction between a group and a team. A team is essentially a group with some special qualities. These include:

- Having good communications and sharing information on a constant basis
- Having respect for each other's abilities and opinions
- Not allowing conflict to become personal
- Having a common goal for which members share responsibility
- Being small enough in size (usually about 10 to 12 people) to interact successfully

Keeping these qualities in mind, you might realize that a lot of groups described as "teams" are not in fact what we should consider a team. This distinction between a group and a team can't be overemphasized when it comes to decision making processes.

As you look at the traits of a team as I just described, think about the impact that each trait has on decision making. Notice also that these traits are related in a lot of ways. If you have the first two traits, good communications and respect for one another, chances are high that you also have the other three traits. But if you don't have the first two traits, it's pretty unlikely that you have the other three.

Back up and look at the examples I have cited so far. When I was an audio guy, the



group of people that decided what music was going to stay in the game was definitely not a team. On the other hand, a small group of programmers working together often will qualify as a team.

Recently I was involved in trying to get a game to final. To do this, we needed the approval of the publisher's testers, the marketing department, the platform manufacturer's focus group, the team that created the original arcade game, and a handful of other people. This is the classic case of a group not being a team. The group didn't share a clear vision of what the goals of the game were, there was poor communication with very little information being shared, and there was little respect between the groups for the opinions of others. Not surprisingly, a number of very poor decisions were made as each subgroup tried to use its influence to dictate the final product. One of the main reasons that so many games fall apart near the end of development is that decision making at this critical juncture is turned over to a group that is not a team.

How can we as an industry try to resolve this problem? In the above case there are two possible scenarios. The first is to pick one subgroup and give them full decision making authority. But which

group? Your answer probably depends on which subgroup you belong to.

Alternatively, we could restructure the way that different subgroups work together, trying to find a way to create a true team. The first objective would be to trim the size of the group down to a manageable team size. One person from each subgroup could be designated to represent that subgroup. These representatives would then have to meet, set goals, communicate, and work together over the life of the project.

This is easier said than done, of course. Such a structure would be a radical departure from the way these subgroups currently interact, and would require that each representative bring a high level of competence to the team. However, this isn't impossible, and "cross-functional teams" are relatively common in other industries from aerospace to waste management.

But the group's ability to function as a team will have a tremendous impact on how successful the group is at making decisions. If the group becomes a team, its members are likely to relate to each other in a cooperative way, trying to sort through the challenges together, while groups that are not teams are focused largely on watching their own backs.

When properly implemented, team/group decision making can have several big benefits. A team can combine its experience and judgment in a constructive way that offers significantly more experience and insight than a single big cheese. Also, when teams work together in a constructive way, they can motivate each other, delivering a product that far exceeds their individual capabilities.

Political Decision Making

Most people are pretty familiar with this method in its most basic form — we decide to do something because someone whom we need to keep happy has suggested that it be done. The decision was made not because the suggestion was a good one (it might have been either

good or bad), but simply because of who suggested it.

Political decision making is much broader than and has lots of variations on the situation I just described. Who hasn't sat in a meeting that dragged on and on until everyone agreed to a specific decision, just so the meeting would end? (This is sometimes referred to as the "he who has the biggest bladder gets his way" strategy.) Similar is the tactic of simply repeating your solution over and over until you wear down all the opposition. In a strict sense, it might be unfair to deem these political decisions, but the key here is that the decision is made because you are trying to appease someone (even if you only want to appease that person enough so that they will let you out of the meeting).

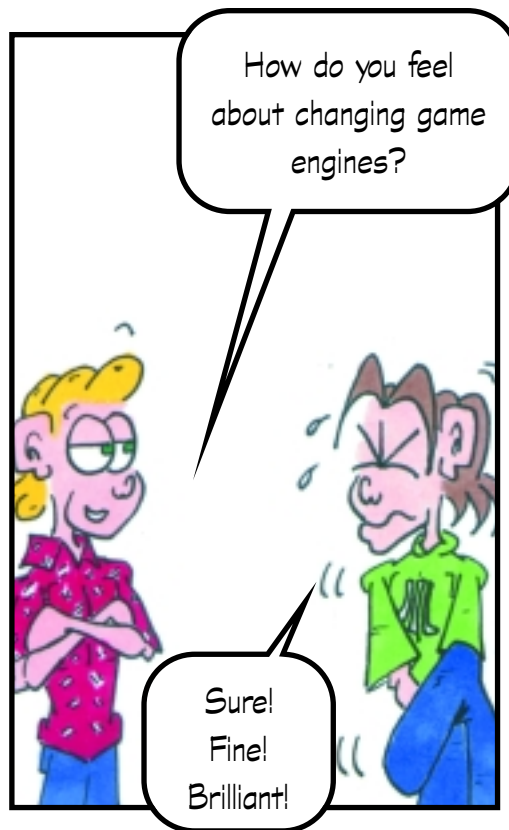
Merit-Based Decision Making

Merit-based decision making involves weighing the possible solutions and picking the best one of the group, based on which solution is best for the game. In this method, different people suggest different solutions and their ideas compete to be the victor.

How you judge the merit of a solution is extremely important, as the quality of your decision will hinge on this. If you judge the merits of the solutions by who proposed them, then you're really making politically based decisions. For merit-based decision making to be effective, you need to have a good, unbiased method of evaluating the strengths and weaknesses of competing solutions. There must be a set of rules or standards to rate the possible choices. Whether a group or an individual is making the decision, these standards should be clearly defined.

How you define these standards is not the issue, so long as everyone shares the same standards. In all the successful teams I've worked on, we've never written the standards down. We had good communications as a team, and we regularly discussed the standards so that everyone understood them.

Consider for a minute the two main tasks of creating a game, programming



The processes you use to make decisions can often change over the course of a project. At the beginning of a project, when you want to encourage experimentation and give people the opportunity to be creative, you might allow the group to make decisions. But as the project approaches completion and you want consistency in design, it might be a lot better to have only one person making decisions.

Likewise, the type of decision can affect what process you should pick. For a group to make good merit-based decisions you need a set of rules or standards by which to judge the choices. Unfortunately there isn't a universal set of rules that will tell you what decision making process to use. You'll need to consider the advantages and disadvantages of the different processes, the specifics of your situation, and what your goals are. If you want to use merit-based decision making, a set of rules or standards that you can use

to evaluate the choices with is an absolute necessity. If a group is making the decision, will the group agree to use those rules, or is it more likely that politics will overrule?

Avoiding Political Decision Making

The curious aspect of political decision making is that most people feel it is a poor way to make decisions in a creative environment. Yet it is surprisingly common. Why is that?

Chefs have an expression, "A fish rots back from its head." This is true for organizations, too. If upper management is political, this will filter down to the rank and file. But if upper management encourages and uses merit-based decision making, then the employees will follow this example. I really don't think this point can be emphasized too heavily: management must set the tone by example. But that isn't all that needs to happen.

From a practical point of view, management must relinquish control of some decisions in order to avoid political decisions. This is often a tough thing for management. At one company I worked for, the art director was in charge of the art, but the CEO

and art creation. From my experience, a team of programmers can come up with a list of standards without much difficulty. But artists have a significantly harder time creating a set of standards that defines what "good art" is. This explains why we often see programming teams making group decisions, but the art decisions are usually made by one person, the art director. That's not to say that an art team couldn't make group decisions based on a set of standards, but the more subjective a decision is, the more difficult it is to make group decisions.

Picking a Decision Making Method

Now that we've identified the basic divisions of decision making processes, let's look closer at deciding which method to use and then how to avoid or promote particular methods. Which method will work best will depend primarily on three factors: where you are in the creative process, what type of decision you need to make (how subjective versus how objective the decision is), and the people in the group (do you have a true team or just a group, and is there anyone who is good "big cheese" material?).

COMMON TRAPS OF GROUP DECISION MAKING

would always come in near the end of the project and make lots of “suggestions.” Knowing the danger of not accepting these “suggestions,” the art director always saw to it that they were implemented. In order to avoid politics, the CEO is going to have to relinquish control.

But it’s not just the CEO. Consider again my previous examples. In a typical game development environment, the development team is given suggestions or requests for changes from the marketing department, the publisher, the license holder, the test department, and the children of the vice president of accounting. If management doesn’t keep these groups at bay, decisions will almost undoubtedly become political. In cases like this, it is crucial that management become proactive in setting up the decision making process.

Other factors can contribute to political decision making, too. When agreeing to a political decision you avoid taking on responsibility. On one game I worked on, I really did not want to implement a feature that had been suggested. But an assistant producer dragged me into an office and wouldn’t let me leave until I agreed to do the implementation — not because he liked the feature, but because it was low-risk to him. If the feature wasn’t well received he could shrug and point his finger at the person who suggested the solution in the first place.

Another common reason to opt for the political decision process is that it is often the easiest path. If you don’t agree to the politically correct solution you might trigger a bloody conflict, or worse, you might not get out of that meeting with your bladder intact! Avoiding political decisions is seldom the easy road. In most cases, it’s up to management to create an environment that does not encourage political decision making.

Management can help discourage this practice in a number of ways. First, give decision making authority to a specific person or group. If it is a group, they should exhibit the signs of a team that I discussed earlier. Second, don’t allow other groups or individuals to interfere with the decision. They should be allowed to make suggestions, but only if they are prepared to have their suggestions discarded. Third, hold the decision makers responsible for the end results. Equally important is for managers

When groups attempt to reach decisions regarding game development, there are some pretty common traps that they can fall into. Here are just some of them:

- **Expecting everyone to reach a consensus.** Whenever a group decides that they will make a decision by consensus, and their anticipation is that a consensus means everyone, they expose themselves to one person holding out until they get their way. To prevent this, the group should be prepared to make a decision without every last person agreeing. A simple majority can often be enough.
- **The solution that offends the least.** When game developers start bad-mouthing “decision by committee,” this is what they usually object to. The best solution is often passed over because someone finds it objectionable. (This happens more often in the case of art and design decisions than programming decisions.) The result is a decision that doesn’t offend, but also doesn’t make for a very exciting game.
- **Too many cooks spoil the broth.** All too often, game development suffers from too many people trying to control the design of a game. Within the company, the development team, the marketing team, the testers, and the executives all feel it is their domain to “fix” whatever they don’t like. Meanwhile, from outside the company we often see the publisher and any license holders also wanting to make changes. The changes requested can be good, but often the suggestions conflict with other aspects of the game or are not thoroughly thought out.
- **Last-minute changes.** This trap usually goes hand in hand with “too many cooks spoil the broth.” Whenever a large number of people or groups want to make changes, they often wait

to tell anyone until the game is almost at final. But late changes can have disastrous consequences. Rather than being designed into the game in a coherent way, they are slapped on piggyback-style. This often introduces new bugs and problems, causing final not to be reached.

- **Letting it get personal.** Members of a strong, effective team never let their differences become personal. If someone attacks your idea, remember that it is your idea that they’re attacking, not you. When developing creative content, the risk of criticism being mistaken as personally directed is very high, so always be alert to the response your comments can generate.
- **No uniformity to the end result.** Whenever you are making group decisions or working as a team developing content, be particularly alert to the potential that your end result will lack uniformity. The clearer your guidelines are, the less likely it is that this will be a problem.



to understand that when the time comes to evaluate the employees, the evaluations must reflect end results, not whether or not the employee agreed with the manager. And finally, management must ensure that people who are not the decision makers relinquish control to those who are.

But what can you do if you aren't in management? This depends a lot on the organization. Unfortunately, if the organization is full of politics, the effect you can have is probably very limited. On a small scale, talking to others in your immediate group about this issue can help. You might not be able to change the organization, but you might be able to affect your immediate situation.

That's the art director.



Encouraging Merit-Based Decision Making

Earlier, I made a distinction between a team and a group. Nowhere does this distinction become more important than when you are trying to encourage group merit-based decision making. If the group is a team, it is usually not too difficult to implement merit-based decisions. But if the group is not a team, you will find it very difficult to get this process to work.

You can encourage successful merit-based decision making by observing a few guidelines. First, create a set of rules or standards that can be used to evaluate the choices. Consider writing them down and putting them in the technical design document. Also, in order to evaluate choices fairly, you need lots of accurate information to base decisions on. Make sure your environment encourages people to exchange and share information. This will help keep decisions merit-based rather than political. Third, be sure to allow everyone the chance to bring their ideas to the table. In meetings, make sure that one or two people don't monopolize the meeting. Finally, be willing to let go of your idea when another idea rates better under the evaluation standards, and never allow any discussion to become personal. Although these rules are meant for group decision making, a single decision maker

can also implement and benefit from any of these suggestions.

Mixing Styles

Keep in mind that you don't need to use only one style of decision making in all situations. For instance, when the project starts you could focus on constructing teams, not groups, and having the teams make merit-based decisions. As

the project draws to an end, the leader then begins to assert more control, and makes more decisions. This scenario requires a leader who is capable of working effectively in both styles, first allowing others to make decisions, and then subsequently taking control. It also requires that the team have enough respect for this leader to relinquish their decision making role as the project comes to the end. On the other hand, if the leader doesn't trust the team's decisions and is thinking, "Oh, don't worry about that, I'll just change that later," you're probably headed for a full-scale disaster.

Last Words

Hopefully in this article I've been able to shine some light on the processes that we use to make decisions. But just raising awareness of these issues is not my objective. This article is meant to be a starting point for people in game development, a set of basic ways to look at what is going on in their organization, which can then be a basis for establishing better decision making processes with better results. I would encourage you to think through some of the consequences and implications of these issues. From my experiences and from the comments of others I know in the game business, we desperately need to look at how we make decisions, and how the outcomes of these decisions affect our companies, our clients, our products, and our customers. 🎮

FOR MORE INFORMATION

BOOKS

- Katzenbach, Jon R., and Douglas K. Smith. *The Wisdom of Teams*. New York: HarperBusiness, 1993.
- Lencioni, Patrick. *Obsessions of an Extraordinary Executive*. San Francisco: Jossey-Bass, 2000.

ARTICLES

- Eisenhardt, Kathleen M., Jean L. Kahwajy, and L.J. Bourgeois III. "How Management Teams Can Have a Good Fight." *Harvard Business Review* Vol. 75 No. 4 (July–August 1997): p. 77.

- Eisenhardt, Kathleen M., Jean L. Kahwajy, and L. J. Bourgeois III. "Conflict and Strategic Choice: How Top Management Teams Disagree." *California Management Review* Vol. 39 No. 2 (Winter 1997): p. 42.
- Katzenbach, Jon R. "The Myth of the Top Management Team." *Harvard Business Review* Vol. 75, No. 6 (November–December 1997): p. 82.
- Morely, Eileen, and Andrew Silver. "A Film Director's Approach to Managing Creativity." *Harvard Business Review* Vol. 55 No. 2 (March–April 1977): p. 59.

Raven Software's STAR TREK: VOYAGER — ELITE FORCE



GAME DATA

NUMBER OF FULL-TIME DEVELOPERS: 20
NUMBER OF CONTRACTORS: 13, including two prerendered animation studios, additional musician, voice director, casting director, and eight main Voyager actors
LENGTH OF DEVELOPMENT: six months of pre-production, 18 months of production
PROJECT SIZE: Single-player and Holomatch: 919,749 lines of code; 1,679 files. The single-player game was largely controlled by scripting, totaling 112,056 lines of code and 2,236 files.
RELEASE DATE: September 20, 2000
INTENDED PLATFORMS: Windows 95/98/NT/2000, Macintosh, Linux, Playstation 2
PROJECT BUDGET: Multi-million-dollar budget
CRITICAL DEVELOPMENT HARDWARE: Average system: Dell Pentium II 550 with 128MB RAM, 18GB hard drive, GeForce 3D acceleration card, and 21-inch monitor.
CRITICAL DEVELOPMENT SOFTWARE: Microsoft Visual C++ 6.0, Microsoft Visual SourceSafe 6.0, Borland JBuilder 3.5, 3D Studio Max 2, Softimage 3D, Photoshop.
NOTABLE TECHNOLOGIES: Licensed the QUAKE 3: ARENA engine from id Software (using OpenGL); Icarus scripting system, BehaveEd scripting tool, Carcass skeletal system, Bink, and motion-capture data from House of Moves.

In the summer of 1998, Activision had acquired licensing rights to make games using a number of *Star Trek* franchises. Their goals from the beginning were to create a broad selection of games and show the gaming community that Activision could take the *Star Trek* brand and make high-quality games with it, better than other publishers had in the past. The preliminary game slate was set with a first-person shooter as one of the initial titles.

Raven Software had been an external studio of Activision for a year, finishing up work on *HERETIC 2* and diving deep into the development of *SOLDIER OF FORTUNE*. *HERETIC 2* was near completion, and we would soon need another project to work on. With our experience developing shooters and a reputation for making quality games, Activision handed the *Star Trek* first-person shooter project to us.

The game started out being based on an unknown *Star Trek* crew within the *Next Generation* franchise. For two months work was done on the plot and story line, with a test level of a Defiant-class ship made using the *QUAKE 2* engine. The main factor in designing the plot of the game was that it had to be an action game, despite the fact that *Star Trek* isn't known for action. To give meaning to the action, the idea for a Special Forces team soon emerged to drive the action for the game. Ultimately, because Activision already had two other games using the *Next Generation* license, the setting for our game changed to the *Voyager* franchise. Our excitement level was low at first, with the team feeling that *Voyager* was the least popular of all the *Star Trek* franchises. We soon realized that *Voyager's* plot allowed us not only to make our game with much more creative freedom, but also to create from something no one else had used. This inspired us to open the floodgates, continue on, and eventually realize that *Voyager* was the best setting for what we wanted to do. We quickly adapted the plot we had at that point into the *Voyager* setting. This was much easier than we thought it would be, and the Elite Force, or the Hazard Team as we called them, actually seemed to make more sense as a by-product of *Voyager's* situation. In January 1999, full production on *ELITE FORCE* began with a small team of 15 people that would grow to about 25 core team members, with additional support from the *SOLDIER OF FORTUNE* team.

Our main focus during production was not to think about the game as a *Star Trek* product per se, but rather an action shooter that borrowed from the *Star Trek* universe. This helped us focus more on what would be fun for players. To our surprise, the Paramount approval process was much easier than we anticipated. We had heard many horror stories regarding Paramount's strictness with their licenses, things like, "You can't do anything new," and, "It's hard to get things approved because they're so protective of the license." What we experienced was the exact opposite. Paramount was more than accommodating in helping us create a fun game, and we were able to bend the rules a little along the way to help accomplish our goal. We created new Starfleet weapons, a *Voyager* SWAT team, used the Klingons, and even added "classic" *Star Trek* to the *Voyager* setting. As long as an element made sense to the story and its presence could be explained, it was no problem.

One of the biggest obstacles we had to overcome was that we would be making an action game that had to appeal to both the hardcore FPS player as well as the average *Star Trek* gamer and fan. This was no easy task, and we spent a lot of time debating over the game style being too much of an action game or more of a *Star Trek* game. Balancing these two aims was a constant battle during the course of production. We knew we had to walk a fine line blending a shooter and a *Star Trek* experience if we were going to both make a successful game and overcome people's perceptions that *Star Trek* games are not good games.



What Went Right

1 ● **Improvements to the QUAKE 3 engine.** Raven had worked with id Software's engines since 1992, but this was the first time we had to add a single-player game to an id engine. Normally, we had the luxury of starting with a full single-player code base and just adding things such as breakable brushes, new AI, navigation systems, and so on. But this time we had licensed a multiplayer game and had to put in many systems we took for granted. We needed AI and navigation appropriate for single-player enemies (not multiplayer bots), as well as teammate non-player characters (NPCs) and cinematics. We needed an

expanded animation system for all the different animations our cinematics would require, we needed to create a load and save routine from scratch, and the list went on. One of the things that made this possible was the decision early on to separate the multiplayer and single-player executables. At this time, QUAKE 3 was still about eight months from completion, so we started on single-player and would worry about multiplayer when we got the final code base. We were able to make drastic changes to the single-player game and shortcut the networking, allowing us to get away with a lot of things that would have just done very bad things to networking. With this new freedom, we revamped even more systems.

In the end, we actually surpassed our initial ambitions as far as new systems and features were concerned.

For example, our Icarus scripting system was planned from the beginning and ended up working out very well. The initial setup was finished relatively quickly and the remainder of the work was mostly just tying the commands to the game and AI. However, for the first seven or eight months, only a couple of programmers were doing any scripting, as they were still refining the commands and there was no GUI for it yet. It wasn't until the fall of 1999 that we made a GUI and the designers could finally start scripting. The system ended up contributing a huge amount to the detail,

BRIAN PELLETIER | Brian has worked at Raven Software for over eight years and was the project leader for STAR TREK: VOYAGER — ELITE FORCE. Besides the two founders, he has worked for Raven longer than any of its 50 employees and has taken on many roles there, from artist to lead artist and managing the art department. His game credits include ten of Raven's games, and he was project leader for HERETIC 2. Contact him at bpelletier@ravensoft.com.

MICHAEL CHANG GUMMELT | Mike is a writer/game programmer at Raven Software. He has worked on various elements of game programming such as AI, navigation, effects, animation, world interactivity, weapons, multiplayer modes, and scripting on HEXEN 2, HEXEN 2: THE PORTAL OF PRAEVUS, SIEGE, HERETIC 2, and STAR TREK: VOYAGER — ELITE FORCE (for which he also wrote the dialogue script). Contact him at mgummelt@ravensoft.com.

JAMES MONROE | James was the lead programmer for STAR TREK: VOYAGER — ELITE FORCE. Starting his career at Origin Systems, he soon moved on to Rogue Entertainment as lead programmer on STRIFE, an action/RPG using the DOOM engine. His Raven credits include programmer on MAGESLAYER and project leader for the HEXEN 2 mission pack, PORTAL OF PRAEVUS. He can be contacted at jmonroe@ravensoft.com.



Extensive photographic reference was used to get as close to the exact likeness as possible for Voyager's interior. Here, the photo reference of the bridge of Voyager is on the left and the game screenshot is on the right.

uniqueness, and complexity of the game, and without it ELITE FORCE would have been a totally different game.

Another big technology decision we had to make was with Carcass, our new skeletal model format. It was a huge undertaking to switch over to the new format, but it really saved us in the end. At first we were using the same model format as QUAKE 3, but it quickly became apparent that we were surpassing that format's capabilities, so we looked for a solution. id had already laid the groundwork for a skeletal model, which seemed like it would work for us. Starting with that basis, we completed it and developed it into the final format we called Carcass. With it, we reduced tenfold the amount of memory a single model took up. Without the Carcass format, we would have had to cut back many animations, and we would have lost the complex detail in our cinematics.

Another technology that was successful was our lip-synch system, which really added realism to the facial animations. We did some research, looking into phoneme recognition, but finally settled on a quick volume analysis. We planned this system to make it very easy to use. Once the mouth animation art was made for each character, the system used the appropriate frame without intervention. The code automatically scanned for peak volume of sounds when loaded, and compared against that whenever a sound was played on the voice channel. Then the animation system picked

up that value to choose the speaking frame. This setup required no extra effort when adding sounds, and would work automatically for any foreign languages used.

Another system we revamped was the cinematic system, which had to be powerful and flexible enough to give our cinematics that *Star Trek* "feel." The camera system itself wasn't that hard to implement, and it worked out well. First, the scripter/designer would set up the blocking of the NPCs through Icarus. Then they could go into the game and let the scripted event play out, pausing it whenever they wanted to save a camera position to a file that could be imported into the map. Using Icarus commands, they could make the camera zoom in and out, change the field of view to simulate close-ups and wide shots, move along a track, dynamically follow a subject, fade in and out, shake, and so on. This allowed us to set up our insane amount of cinematics as quickly as possible and still allow for some fine-tuning and detail (such as the "walk and talk" with Tuvok and Munro, and especially all the gestures and expressions the NPCs themselves would do to add to their characterization and the believability of a scene).

2 ● Complete plot and story right from the start.

From the beginning of production, ELITE FORCE had a detailed story line, and every level of the game was written out in story form. We also had standards we had to meet; after

all, our game was going to be compared to the *Voyager* TV show, so there was even more emphasis on storytelling. The story had to be engaging and reminiscent of what a *Voyager* episode would be, and we had to make sure our story had a lot of depth and interest for the player, to give them the feeling that they were partaking in an episode of the show. Since one of our main goals was to have an away-team accompany the player for the missions, it was even more important that the story be solidified up front. A lot of story is conveyed during the missions, so we had to make sure the levels were paced out well and the level designers knew up front what story content their levels contained. We were able to pace the story throughout the game so that players would be continually rewarded with exposition. As they completed more missions, the overarching plot of the game would slowly form in their minds.

With our tight schedule, we wouldn't have time to redo parts of the game if they didn't work out, so it was crucial for all the people involved to work together on the story line toward one common goal. With a complete walkthrough of the levels written, the level designers could concentrate on the looks of the level and accommodate for where the cinematics and story segments were going to take place. Because our story line never changed during production, we were able to proceed forward uninterrupted, and



ABOVE. Storyboards were created as guides for the in-game cinematics, helping to speed up production of more than 50 cinematic sequences.

never had to scrap any levels due to plot restructure. This was key, as we had a fairly small team charged with creating a lot of content in a short period of time. The majority of the dialogue was written after all the levels were finished, but this too went smoothly because the walkthroughs were updated from the finalized levels, and then the dialogue was written from them.

3. The dialogue. The team was excited about the concept of playing with a team of NPCs in an FPS. This led to the definition of the different characters of the Hazard Team early on. However, the actual script for the game didn't begin right away, since that had to wait for the final game flow design to be finished. Our writer (also one of our programmers) started in September 1999 and finished the first draft in March 2000.

While he was writing the script, we were making all the cinematics and needed some temporary voices. We had employees record the lines and then dropped them into the cinematics to give us a feel for the pacing of each scene. This allowed us to tweak the dialogue while saving us from having

to bring the expensive *Voyager* cast back for pickup lines.

The script finally came together after many revisions and, once it was approved by Paramount, the actors were lined up very quickly and the voice recording was done in about a month. We were then able to put the final lines into each scene, replacing our temporary dialogue without having to adjust the timing or change the scripts. This was due to the fact that *Icarus* could pause execution of a script until a sound file finished playing, so dropping in a new line of dialogue automatically adjusted the timing of each scene. This also meant that lines would generally flow better in translation, since they wouldn't have

to deliver the line too quickly or slowly to match any hard-coded timing.

We also had a system for automatically reading the dialogue script itself and turning it into .PRE files that would let the game precache all the dialogue for a level and simultaneously assign the caption text to it. Included in this was automatic localization and dialogue adjustment for the player's gender. All of these things together enabled our game to have captioning, localization, gender-specific dialogue, precaching, lip-synching, and almost no pickups.

In the end, the dialogue turned out very well and our performances were good (Paramount even let us make final casting



RIGHT. *Voyager*'s Hazard Team, created by Raven to help accentuate the action for the game.



ABOVE. Storyboards also referenced the game's dialogue script, which was more than 400 pages long — twice the length of a movie script.

decisions on all the ELITE FORCE-specific characters). The story and dialogue added a great deal to the game and contributed heavily to the feeling of actually being in an episode of *Star Trek*.

4 ● **Re-creating the look of the *Star Trek* universe.** Raven has always tried to push graphics boundaries and painstakingly create beautiful settings for our game worlds. *Star Trek* was no exception, and challenged us not only to create a beautiful gaming environment but also to create it in a likeness that is known worldwide. It's one thing to make arbitrary-looking levels in a never-before-seen world, but when trying to re-create the look of *Voyager* we came across many difficulties that we hadn't expected. For

starters, the *QUAKE 3* level editor is made to create levels at a fairly big scale. When we built the bridge of *Voyager*, we were all astounded by the detail that we achieved, but when we put a normal-sized character on the bridge, he was incredibly tiny. The bridge was huge, yet it was built like you would build any normal *QUAKE 3* level. We realized that we had to build the levels on a much smaller scale than what we were used to. It took seven attempts at rebuilding *Voyager's* bridge until we attained the proper proportions between the characters and the level. We tweaked the scale until it looked perfect and the player and other characters could move around with ease. Once the scale of the levels was set as a standard, we continued forward with the other *Voyager* rooms with little rework needed.

The artists spent much time working on the textures for the environments, getting their reference from many sources to make sure everything was exact. Having access to Paramount's *Star Trek* reference library was key in getting reference for carpets, chairs, upholstery, computer panels, and more. We sometimes scanned in the photo references themselves for the textures. Watching episodes of the show on tape was also instrumental in determining what things looked like. We used a total of 1,033 textures to create the look of *Voyager's* rooms and hallways. Working together, the level designers and artists created the best-looking *Star Trek* environments in any game to date.

Just like the challenges we faced building the environments, creating *Voyager's*



ABOVE. Menu screen for choosing your player character in Holomatch gameplay.



ABOVE. Fighting aliens in the stasis ship, which was created with 90 percent curved surfaces. **TOP RIGHT.** Save your teammate from the Borg, one of the many multiple outcome events. **BOTTOM RIGHT.** Your teammates fighting alongside you, with Telsia blasting Etherians.

characters and crew presented the challenge of re-creating something that already existed. We used many references from the *Star Trek* reference library to help us re-create these real people. Each actor had a series of photos of him or her as their character, which we used to help get just the right shape of the head, and we even used the photos themselves (with Photoshop touch-ups) as textures on the polygon heads to achieve the likenesses. There are limitations to any technology, and working around the limitations is where we succeeded. The heads could only have 150 polygonal faces and the textures for the skins could only be 512×512 pixels. The fine craftsmanship required to work with so little and still achieve the right look for the characters is a testament to our artists' skills. For example, designing the Hazard Suit in the *Voyager* style took many attempts, but we finally got something everyone was happy with and looked natural to *Star Trek*.

5 • Creating smart NPCs. To create a *Voyager* game that resembles what you would see in a TV episode, we had to create a working spaceship filled with its busy crew, and make it believable enough so players would feel like they are

in a real place. We also had to create an away-team to fight alongside the player. After all, what is *Star Trek* without an away-team?

We created our NPCs using a few different things. The Icarus scripting system allowed us to have precise control over specific actions. The NPC Stats system allowed us to create many characters with various looks. The Squadmate system gave us the tools to enable the teammates to work with the player, and we used a waypoint or pathfinding system to make our NPCs navigate through a complex environment. All of these systems together created the artificial intelligence for our NPCs. We used scripting with the pathfinding to make the crew of *Voyager* come to life, and we could tell them exactly where to go and what to do without too many unknowns to cause problems. They did exactly what we as designers wanted them to do.

The teammates, on the other hand, have to act according to what is happening with the player. Since players can do anything they want while playing the game, this means a lot of unknowns for how the teammates should react. The teammates could have easily been a hindrance to the gameplay, and now we know why no

other company has ever tried to make an FPS game with up to five teammates working alongside you throughout entire levels. In the final stages of developing the teammates, we weren't sure if they were going to work out; they had so many problems, and every time we would fix something another problem would crop up.

Just getting them to follow you was no easy task and was something we kept tweaking right up to the final days. Sure, we could get them to follow the player, but the game took place in tight hallways and small rooms so players would bump into them. They wouldn't get out of the way and would constantly get stuck on each other. Also, having them follow the player everywhere made them seem less like intelligent characters, so sometimes we had them stand their ground or take up a position while the player went exploring. Then we had the constant problem of the team not following players at all, even though they might need them later on. When we did get it working, someone found a new way to break a level with a teammate.

Elevators and teleporters added to the risk of teammates being left behind. We were getting worried that we wouldn't even be able to get them to walk through



TOP LEFT. The AI of the Borg had them adapting to your weapons like their TV show counterparts. **BOTTOM LEFT.** Exploding architecture helped in strategizing your attacks. **RIGHT.** The Klingons' AI allowed them to crouch and run for cover.

an entire level, and we would have to resort to something drastic. Luckily, we did get them to work in the levels. They may have run funny or jumped down long elevator shafts to catch up with the player, but at least they stayed in formation through the whole level no matter what the player was doing.

Of course, once there were enemies we had to worry about friendly fire. We wanted the team to react intelligently to being shot, but we didn't want to punish the player for shooting them accidentally. After a lot of trial and error, we decided that teammates would retaliate against a player only if the player had shot them repeatedly outside combat. In combat, they'd still react to friendly fire, but couldn't be killed, and would never turn on the player.

Then came the problem of trying to balance the teammates' involvement in combat. Once we put enemies in the levels, we found that the teammates were so good that they killed most of them, leaving little for the player to do. To balance this, we had the teammates shoot less often, but then they got attacked constantly by enemies, turning the gameplay into "shoot the aliens attacking your teammates." Eventually we made the ene-

mies attack the player more, so the player would feel threatened by them, and the teammates helped but didn't do all the work. It's funny now to hear people say that the teammates were stupid because they hardly killed any enemies, or that the enemies were dumb because they attacked the player more than the teammates. If they only knew how tiresome the gameplay would have been had we not balanced it the way we did.

What Went Wrong

1. Not enough programmers. For the first half of development, there was mainly only one programmer working on scripting, enemy AI, teammate AI, pathfinding, and the animation system. This programmer was also writing all the dialogue, and it became necessary for him to relinquish other programming duties in order to finish writing. Unfortunately, we didn't have any extra programmers to help.

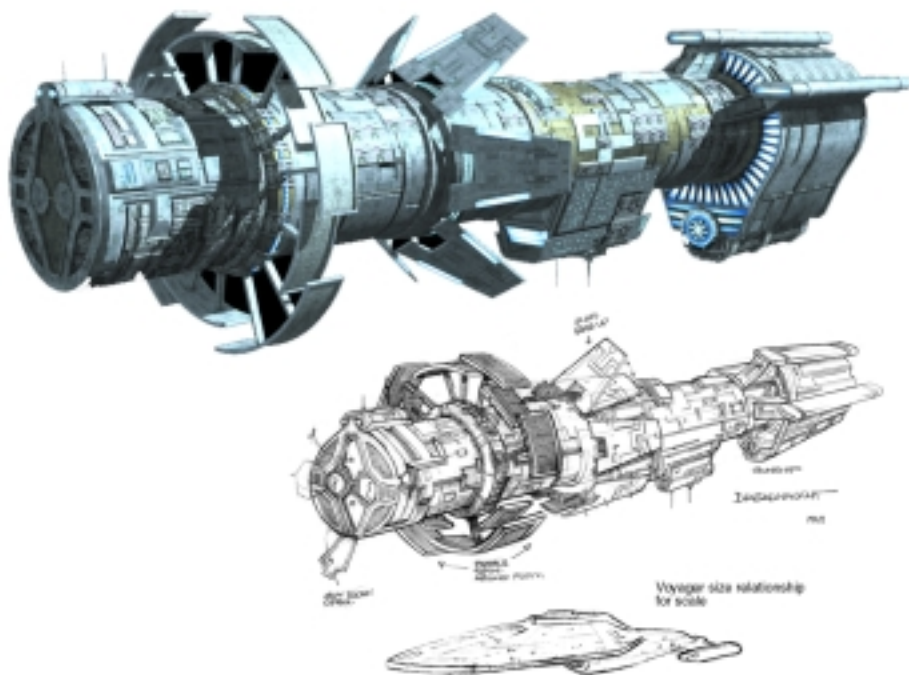
Eventually we got a programmer from a different project to start working on game code. He completely rewrote the navigation system, which took time away from creating the AI for all the enemies, which didn't end up being completed until the game itself was done. This creat-

ed a real lack of cooperation between level design and enemy AI, and forced the designers to rewrite their scripts constantly to match the changes in the underlying game systems. With more programmers working on AI and navigation early on in the development, these kinds of last-minute changes and back-end design could have (hopefully) been avoided.

2. From Ghouls models to regular models to skeletal models.

It was a big decision to switch to the new skeletal models. In the beginning, we were using what was to become SOLDIER OF FORTUNE's Ghouls system (see "Raven Software's SOLDIER OF FORTUNE," Post-mortem, September 1999, for more on Ghouls). When we received the QUAKE 3 code, we tried to integrate Ghouls into the new code, but found it to be too different. It just didn't fit in with the new optimized rendering pipeline QUAKE 3 provided. So we switched over to regular QUAKE 3 models.

There was a lot of learning going on at this time. When we get new code, it doesn't come with operating instructions, and it's often not complete. We went through a lot of growing pains adopting the new format and figuring out its requirements. When



TOP LEFT. The Voyager 3D model used in prerendered cinematics is the actual one used for the TV show. **BOTTOM LEFT.** Other models were designed by Raven and made by the company making the prerendered movies. **RIGHT.** It was important to make highly detailed sketches, since another company was making the models from them.

the option to go skeletal came up, we had to weigh the benefits of the new system versus the risks and time it would take to switch over. While we did the right thing and embraced the new technology, we had to write a new set of tools to handle the new formats and learn new procedures to get our animations out of Softimage 3D and into 3D Studio Max. We had a lot of squashed creatures and bizarrely stretched limbs along the way, but it was well worth the trip. Unfortunately, by the time we got it working right, we were past our alpha date, and because of all the different changes the models had gone through by that time, we didn't have enough time to fully implement a good AI system for the enemies, and had to settle with what we could do in the time we had.

3 • Underestimating the amount of scripting work needed. As we mentioned previously, our Icarus scripting system was a huge plus. But we also encountered a lot of problems with scripting.

Not only had no one ever used this system before, none of the programmers be-

hind it had ever written a scripting system before. The designers didn't even get their hands on the scripting system until about eight months before the game was done. They did pick it up quickly, but not without a lot of effort and time involved.

One of the major problems designers had when scripting was the constant changes to the underlying systems that the Icarus commands relied upon. They'd script an NPC one way and it would work fine. Then the following week, something in the navigation, AI, or animation systems would change, and the script would be broken. This was a source of major frustration among the designers and definitely impeded their productivity. Ideally, those systems would have been finalized before the designers had to start scripting.

Within Icarus itself, there was one major flaw that should be addressed. Icarus can start a command and wait for completion, but it does not have a built-in system for letting the command (or "task") time-out and continue or take another route. There was no failsafe if, somehow, a command never completed. Given the sheer complexity of our scripting, these kinds of showstopper problems showed up constantly and

would completely stop the game in its tracks. Up until the last minute, we were frantically trying to find every case in which a script would just stop execution. In the end, we did manage to catch them all, except for two cases that were caused by people turning their detail levels up too high and causing the game to drop to very low framerates, which could in turn mess up the scripted events. It turned out pretty well in the end, but all the effort that went into constantly revising the scripts could have been put to other, more productive uses.

4 • Adjusting to new QUAKE 3 technology. The biggest level design headache in working with technology that is still being developed is that it's constantly changing. We started building our levels way before the QUAKE 3 engine was near completion, and this caused scheduling problems every time we got a new code build. We built our levels one way with the tools and knowledge we had at the time, and then when a big change was made to the QUAKE 3 code, the level designers had to spend a few days altering each of their levels to keep

up with the changes in the code and how it handled surfaces, lights, and architecture. This happened numerous times during development, and we often went months without new code drops. The level designers would continue to work on levels in order to make progress, and then when we finally received new code, we had to go back to all the levels that had been done and spend a month getting them up to date. This month was not accounted for in the schedule, and therefore a month of designer time was simply lost. This happened more than once and was a big factor in keeping us behind our original schedule.

Another part of adjusting to the QUAKE 3 technology was realizing that our levels couldn't be as big as QUAKE 2 levels. The wonderful thing about the QUAKE 3 engine is that you can have many more polygons in the view at one time, allowing for more detailed levels and rooms, and of course curves. Without this engine we wouldn't have been able to create such accurate-looking *Voyager* rooms and locations.

When we started building levels, we made them as we did using QUAKE 1 and 2 technology. We had expansive levels that looked awesome and showed off what the QUAKE 3 engine could do. Then, somewhere near the middle of our development, we realized that the file size of most of our levels was huge, running 11 to 15MB each, when they should have been about 6MB. This was a problem, since we'd planned for the file sizes to be 10MB or less out of a total memory budget of 64MB. The levels were the normal game-world size of a QUAKE 2 level, so what was the problem?

It turned out to be the high polygon (or triangle) count used to create a much more intricate and detailed environment. We realized that although QUAKE 3 can handle more polygons in the view at one time, the file size for the level had not increased much from a QUAKE 2 level. We had a dilemma; either we could bring the file size down by taking out all of the detail that made the QUAKE 3 engine superior and keep the physical size, or we could cut the size of the level down, making it smaller yet highly detailed. Since we were making a world that could readily be compared to a TV show, we opted to keep the detailed environments of the *Star Trek* universe and cut the level size down.

We were able to cut most of the levels in half and make two separate levels out of them, but then all the level designers had twice as many levels to work on, and this could have caused some major scheduling problems. Unfortunately, to keep up with the schedule, large parts of the levels were deleted and redesigned, which resulted in much smaller levels that could be traversed quicker, and ultimately made for a shorter game.

5. Mission stats never got finished.

The only major thing that didn't get into the game that we originally planned for was our end-mission statistics. The feature made it into the game in the form of basic stats when you died, but it was planned to be much more, and would have really added to the game. The end-mission stats would have improved the replayability of the single-player game and given more emphasis to the interactive and multiple-outcome events. The main goal with the stats was to grade players' performance when they completed a mission; for example, giving a score of 100 percent for a perfect mission. A number of medals would also be awarded to players based on what they did during the mission. At the end of the game, all the scores and medals would be added up for one final game score.

The stats would have made a significant gameplay feature, letting players know at the end of a mission whether they could have saved someone or done something differently to get a better score. This would have made the interactive events mean something more, emphasizing the fact that they *are* interactive; events that players didn't even know could be changed would have been presented to them after the mission, making the game world seem that much more alive. With this feature not in the game, the multiple outcome events didn't mean anything more than just a "cool" factor, instead of being intrinsic to the gameplay and adding to replayability.

Final Thoughts

We started out with a lot of great ideas, and almost all of those ideas were implemented in the game with the exception of a couple. That's certainly an achievement in this industry.

We made the game we set out to make and are very happy with the end result, so it was hard to think of the things that went wrong. Even though we had many problems, we worked around them and ultimately finished the game, which makes us feel like we did everything right. Then we heard comments from fans and game reviewers who didn't like certain things about the game, and all the memories of working through all the obstacles came back, and we thought, "If they only knew how many problems we had to work through to get the game to its final state." It's working through those development obstacles that makes a game successful. It's also gratifying to hear from fans and reviewers that the game was successful both as a fun game and as a *Star Trek* game. For us, that means many more things went right than wrong, and the team was talented enough to work through the things that went wrong and make a game that is being enjoyed by thousands of people.

ELITE FORCE was a very difficult project cycle with a really long crunch mode, but what game is any different? Yet we had a lot of extra obstacles to overcome, including the perception that all *Star Trek* games suck and that meant ours would too. It seemed like we were destined to fail before we even started. Working with one of the two biggest science-fiction franchises in

the world added to the pressure. But

Activision supported us all the way from upper management to a top-notch marketing and PR staff. Paramount was surprisingly helpful, and proved to us that they care about the quality of the games made and would do everything possible to ensure that quality.

With a dedicated and very talented team of individuals, we met our challenges and succeeded in making what is being called the best *Star Trek* game ever made. 🙌



Viva la Différence

Making the Most of the War Between Artists and Programmers

How do you make a great game without killing each other? You're a talented artist, who is creative, social, and has a thorough knowledge of the latest technologies used to make games. You're working with a talented programmer who never comes out of his room, wants to rewrite everyone's code, and doesn't interact well with others on the team. The hours are long, the technology is changing faster than you can buy it, and you're working on opposite sides of the brain. With the rising demands of the game industry and the next-generation platforms, the team concept is crucial for game development. Yet here are individuals who are complete opposites of each other, trying to work together toward a common goal — to ship a great game.

Getting off on the right foot. At the beginning of a project, artists are developing the look and feel of the game, while the programmers are working on the engine as well as tools that artists will need to implement their shells, shape building, level building, animations, and special effects. It's during this phase of production where things can and usually do go wrong. There is a tremendous amount of pressure on the artists to get the project off on the right foot. If there is no interaction between the artist and the programmers during this period, tons of

art files will be thrown away. Many programmers are not by nature social or interactive, so the artist must knock down the barriers of communication to find out what may or may not be the proper procedure when developing poly count or bitmap size. During this period, it's common for programmers to assume huge amounts of art will be thrown away. The end result is a fair amount of finger pointing and a large amount of time and money lost.

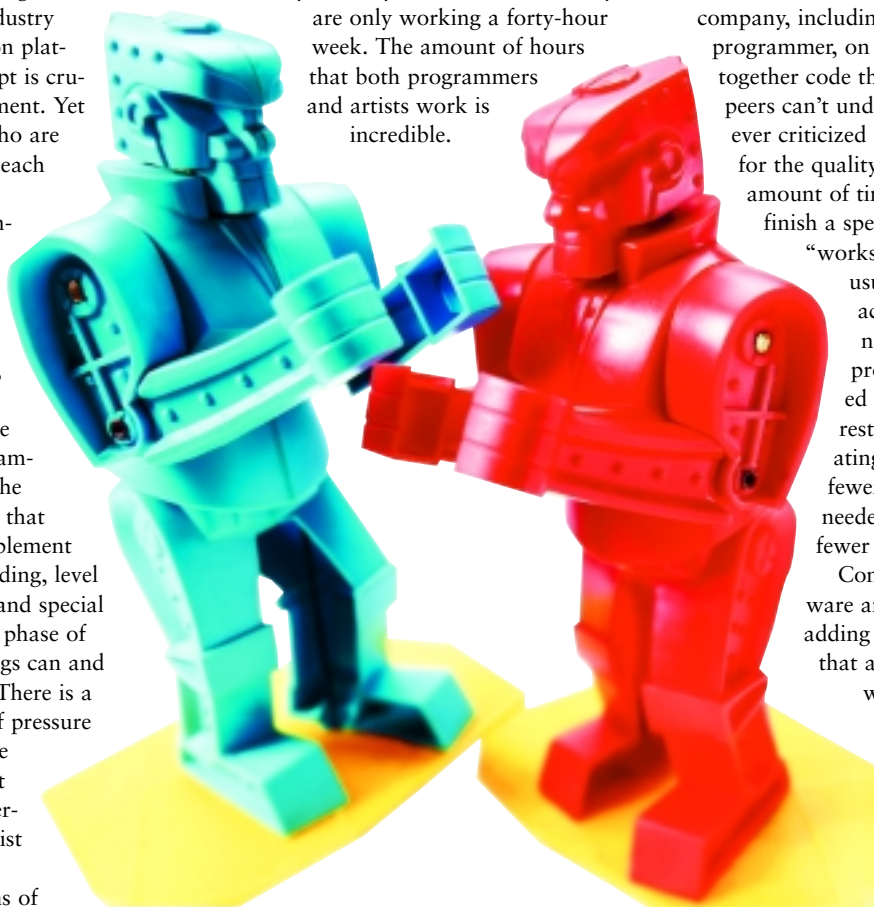
Input leads to output. The norm in the industry is to work flexible hours. This usually means you should feel bad if you are only working a forty-hour week. The amount of hours that both programmers and artists work is incredible.

The problem is that most programmers tend to come in late in the day and work late at night. Many even come up with the lame, "I was working at home last night until three in the morning." They don't feel the need to interact with other team members, even though they are building the very tool that someone else is going to use to implement features. Without advice from designers or artists, many tools end up being totally useless, and not used for their intended purpose. An artist has to visually produce a large body of work that can be criticized by every member of the company, including the receptionist. The programmer, on the other hand, can put together code that sometimes even his peers can't understand, and is hardly ever criticized by other team members for the quality of the tool or the amount of time it has taken him to finish a specific tool. Most tools are

"works in progress," and are usually rewritten later to accommodate the exact needs of the game. If the programmers communicated more effectively with the rest of their team while creating the tool, however, fewer rewrites would be needed, and cleaner code with fewer bugs would be the rule.

Constantly changing software and hardware are also adding to the amount of hours that artists and programmers work. To learn new software, artists at most companies are required to put in extra hours of their own time. Learning entirely

continued on page 63



continued from page 64

new software is only half the problem, however. Artists spend a large amount of time adjusting to new interfaces on upgrades of software they had previously mastered. Furthermore, many artists are beginning to take scripting classes to use with programs such as 3D Studio Max. Changes in software and hardware likewise put added pressure on the programmers to come up with new and fancier game engines and plug-ins. Next-generation platforms seem to be more artist-friendly with higher poly counts, higher resolutions with 24-bit color, and everything in real time.

The constant with even the new platforms, however, is that the artist and the programmer still need to work together toward common goals. To build art tools, exporters, terrain editors, and the like, it takes input from both the artist and the programmer to make the proper tools. There may be a handful of individuals out in the gaming world who have top-notch art, math, design, and programming skills all rolled into one, but I haven't met that person yet. It would be great to hire art techs who could do it all, but I'm not sure how a team of individuals with the same exact skills would work together. A greater variety of artists with different skills can only improve the outcome of a final product. It is much the same with programmers. The more expertise a programming team has with combined 3D, AI, physics, tool-building, and so on, the higher the

quality of their game. People with overly similar skill sets tend to let their egos get in the way of their interactions with the rest of the team, and the results are games delivered late and of lesser quality.

Leveling the playing field. It's no secret that, on average, most artists are well underpaid compared to their programmer counterparts. This is a fact, even though most artists in the industry have college degrees or some kind of art training before they are hired. The gaming industry has far and away rewarded programmers with much higher wages than artists. Many artists I have worked with have become designers just because of the higher salary they could earn. One producer summed it up best by saying, "To get a good artist nowadays, all you have to do is kick a tree and one will fall out."

Every year since I began working in this industry, art has become more and more the critical path. The skills needed to work for a top game company have increased tenfold. More and more artists are learning to script, doing lower-level programming, taking film classes, and so on. Artists are becoming more tech-oriented in part to play a larger role in the creation of a game and thus increase their earning potential. This discrepancy in wages also plays a huge role in the interaction between the artists and the programmers. It often seems that because the programmer is the higher-paid employee, he or she doesn't have to ask the artist for advice or direction. The reality is of course that the best tools and games are created by

individuals who interact together, take direction well, leave their egos at home, and who are rewarded equally for their time.

I have worked with some of the best programmers in the game industry. By far the best programmers to work with were not the ones thought to have the cleanest code, but the so-called hackers who could make the game work and look good. These were also the programmers who worked the fastest and were fun to work with. They work long hours, are social enough that you can interact with them, and come up with great product. Some of the best artists I have worked with are not "techies," but know enough about certain tools to create great-looking game graphics. It does, however, take a variety of kinds of people to make a great game. Clean code makes for a good foundation. Creative, innovative, and talented artists and animators help make that code come to life, good game design and scheduling finishes the job, and if you're lucky everybody lives to ship the game. 🍀

JARRETT JESTER | *Jarrett is art director for Pipeworks Software Inc. and has been in the computer game industry for ten years. Before that, he was an art director at Dynamix, a Sierra company, in their game simulation department, and worked on such titles as ACES OF THE DEEP (a submarine simulation), RED BARON II, and CURSE YOU! RED BARON. He also worked on ACES OF THE PACIFIC, ACES OVER EUROPE, RED BARON, and a variety of adventure games.*
