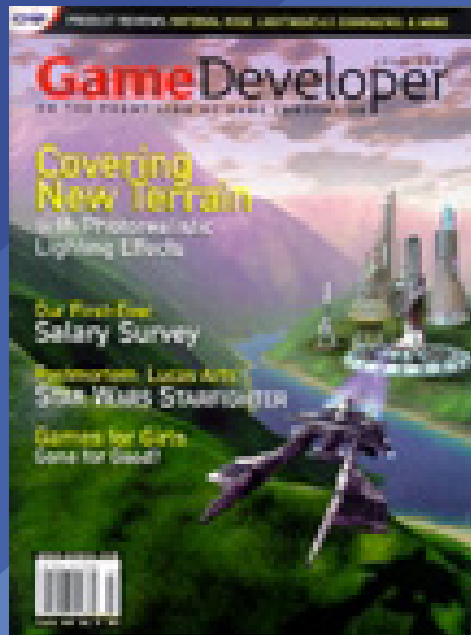




GAME DEVELOPER MAGAZINE

JULY 2001





GAME PLAN

LETTER FROM THE EDITOR

What Are You Worth?

Why are you a game developer? Developers of games typically make less money than developers in other industry sectors. There must be something beyond salary driving us, for a group of people so well-versed in the circadian rhythm of the human body during crunch time. Perhaps it is the simple satisfaction of creating something that you can point to on the shelf and say, "I made that!" Or it could be the power of sharing a story or experience with others. Maybe you enjoy creating technology that looks and plays better than anything else available.

Regardless, receiving more money pretty much always makes a person a little happier. But how do you know that you're getting paid what you're worth? If you took a different job, would you get paid more? What if you moved to a different area? It's challenging to find out this information given how secretive people are about their income.

One of the things we're about here at *Game Developer* is making your jobs easier. We'd like to think that the information we provide you reduces your crunch time, and makes your game look, sound, and play better when it hits the shelves. So we set out to find you some solid information about your salary opportunities. This month's salary survey combines the data we've accumulated from *Game Developer* magazine readers, Gamasutra.com members, and Game Developers Conference attendees over the past year. We've taken a bunch of cuts at this data, separating it out by job title, location, years of experience, and other categories.

We expect this information to be useful for those of you who are experienced developers — after all, people in this industry generally change jobs about every two years, so you'll be interested in checking out how you might be able to make more money on your next move. But this data is also really useful for people who are thinking about joining our industry, so we've included some information about what game development com-

panies are really looking for these days, and how best to maneuver yourself in your career.

Breaking In

Schools for game development are becoming a strong force in our industry. Between the schools and the amount of information available on the Internet, it is much easier to get an education in game development than it used to be. And the demand is high. Just take a look at the number of schools (listed on the IGDA web page www.igda.org/schools.htm) which currently have some game-development-related curriculum: 11 schools in Canada, nine in Europe, one in Asia, 38 in the United States. Some of these schools are completely dedicated to game development, while others provide a few courses in game programming, interactive storytelling, and so on.

While developers with a few titles under their belt will always be the most valuable, it's only recently that graduates from these programs have become important. A few years ago we used to scoff at new graduates — they would come into game development and attempt to use advanced software engineering practices, or design general-purpose C++ classes, or create gorgeous 3D models out of curved surfaces. These days this stuff is actually useful. We used to be worried about where we were going to get another thousand polygons per second out of our graphics engines; now we are pushing the state of the art in technology with our titles, so all that college research and rigor is immensely valuable.

With the massive size of modern PC and console titles these days, it's important to examine new techniques which can help us be more efficient day-to-day. Although it may be a lot of fun to hack something out quickly, that doesn't necessarily work so well when there are 50 programmers and artists hacking on the same game.

GameDeveloper

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

Publisher
Jennifer Pahlka jpahlka@cmp.com

EDITORIAL

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

Senior Editor
Jennifer Olsen jolsen@cmp.com

Managing Editor
Laura Huber lhuber@cmp.com

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

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

Contributing Editors
Daniel Huebner dan@gamasutra.com
Jeff Lander jeff@darwin3d.com
Tito Pagán tpagan@w-link.net

Advisory Board

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

ADVERTISING SALES

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

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

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

Account Manager, Recruitment
Morgan Browning mbrowning@cmp.com t: 415.947.6225

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

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

Sales Associate
Aaron Murawski amurawski@cmp.com t: 415.947.6227

ADVERTISING PRODUCTION

Vice President/Manufacturing Bill Amstutz
Advertising Production Coordinator Kevin Chanel
Reprints Stella Valdez t: 916.983.6971

GAMA NETWORK MARKETING

Senior MarCom Manager Jennifer McLean
Marketing Coordinator Scott Lyon
Audience Development Coordinator Jessica Shultz



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
Executive Vice President & CFO John Day
President, Business Technology Group Adam K. Marder
President, Specialized Technologies Group Regina Starr Ridley
President, Technology Solutions Group Robert Faletta
President, Electronics Group Steve Weitzner
Senior Vice President, Human Resources & Communications Leah Landro
Senior Vice President, Global Sales & Marketing Bill Howard
Senior Vice President, Business Development Vittoria Borazio
Vice President & General Counsel Sandra Grayson
Vice President, Creative Technologies Philip Chapnick

GamaNetwork
A DIVISION OF CMP MEDIA LLC

WWW.GAMANETWORK.COM

THE FORUM FOR YOUR POINT OF VIEW. GIVE US YOUR FEEDBACK...

Author Stunned at "Impulsive" Editorial

I'm amazed that the editor-in-chief of *Game Developer* would pen an editorial ("Violence and Education," Game Plan, May 2001) with a statement as absurd as the following: "One thing that a violent game can teach a person is how to be a sharpshooter. Those arcade games with guns — they're great training devices for learning how to hit targets quickly and accurately." That certainly seems to be a popular theory today, Mr. DeLoura, but I'm afraid that it's wholly untrue. If coin-op shooters, "those arcade games with the guns," provide any advantage for the individuals who play them at all, it involves improving their motor reflexes or hand and eye coordination.

The current craze for suggesting that computer games are somehow responsible for producing violent "sharpshooters" is nothing new; rock-and-roll, comic books, horror movies, cartoons, heavy metal, Dungeons and Dragons, and Elvis are but a few of the things that have been accused of complicity in whatever happened to be the most notorious social problem of their day. With computer games (now as with these before), the argument that exposure to something in any way diminishes an individual's responsibility for their actions is ridiculous. I don't care if Ozzy Osbourne personally sneaks into your room each night to whisper into your ear his unflinch-

ing support for any plans that involve pushing Mom down the stairs, if you actually do it, it's still exclusively your fault, even if he is really persuasive.

Ian M. Fischer
Ensemble Studios
via e-mail

MARK DELOURA RESPONDS: *Ian, I believe you've misinterpreted what I stated. I didn't say that "computer games are somehow responsible for producing violent sharpshooters," nor did I say that "exposure to something in any way diminishes an individual's responsibility for their actions." A game that rewards you for pointing at a target and pulling a trigger doesn't teach you to be violent, and it certainly doesn't diminish any responsibility you have for your actions if you choose to be violent, but you can't argue that it doesn't teach you how to point and shoot. It may not teach you how to shoot accurately, but it certainly helps. It's just as important for us to be clear about the impact our games do have on people as it is for us to be clear about the impact they do not. I trust that you feel the same way.*

Games Can Create Positive Change

Some good points were made in your editorial "Violence and Education." Unfortunately, we don't know whether vio-

lent games affect people's behavior. But here's one indirect piece of evidence: Click Health (www.clickhealth.com) gave a group of kids with diabetes PACKY & MARLON, a game that teaches kids about managing their diabetes. Here's some information from their web site: Children and teens with diabetes who voluntarily used PACKY & MARLON at home for six months reduced their unscheduled urgent care visits related to diabetes by 77 percent compared to the control group. This represents a reduction from 2.4 urgent visits per child per year down to 0.5 urgent visits per child per year. The control group did not change in their rate of urgent care visits.

If health games can have such a dramatic effect on kids' behavior, it seems likely that they're learning something from violent games as well. So what do we do about it? I don't believe in government regulation, but as your editorial stated, we can develop games with positive messages to provide alternatives for kids. In my own work, I'm developing a 3D game to teach math to high school students called AQUAMOOSE, located at www.cc.gatech.edu/elc/aquamoose.

Amy Bruckman
Assistant Professor, College of Computing
Georgia Institute of Technology
via e-mail

Send e-mail to editors@gdmag.com, or write to *Game Developer*, 600 Harrison St., San Francisco, CA 94107

Kludge by Tiger Byrd and Daniel Huebner



INDUSTRY WATCH

daniel huebner | THE BUZZ ABOUT THE GAME BIZ



3D patent awarded. The United States Patent and Trademark Office has awarded a patent covering scalable 3D server technology. 3D virtual reality entertainment portal Worlds.com has been granted U.S. Patent #6,219,045 for its scalable architecture for a three-dimensional, multi-user, interactive virtual world system. The technology covered by the patent is described as allowing multiple users to interact in a three-dimensional, computer-generated graphical space where each user executes a client process to view a virtual world from the perspective of that user, with avatars representing the other users in the virtual world and the user's view updating to reflect the motion of other users by way of a central server processor that provides position updates to client processes. The client process also employs an environment database to determine which background objects to render as well as to limit the number of displayable avatars to a maximum number displayable by that client. Not unexpectedly, Worlds.com believes that its patent may apply to currently existing multiplayer games and has indicated its intention to review offerings that may be infringing on its new patent.

Sega makes cuts. Sega made plans to slash its workforce by 28 percent in a bid to return to profitability within a year. In addition to cutting its payroll, the company will sell off a number of the 58 companies it currently owns. Around 200 of the job cuts will be made through a voluntary retirement plan, leaving a balance of about 180 people that will be affected. All of the planned cuts will be made at Sega's Tokyo headquarters, taking the company from its peak of 1,081 staff in January to a target of just 700 by March 2002. Sega's total investment in outside companies currently stands at about \$115 million; the company will base its divestiture strategy upon whether the companies are important to Sega's business and the degree to which they are profitable.

Interplay's revenues climb. Interplay has reported improved operating results for its fiscal year 2000. Net revenues climbed



Worlds.com's scalable 3D server patent may affect 3D multiplayer online games such as TRIBES 2.

to \$104.6 million, from last year's mark of \$101.9 million. That modest gain helped Interplay cut its net loss for the year to \$12.1 million from last year's \$32.8 million. Interplay still realized close to 75 percent of its fiscal 2000 revenues from PC titles and is taking measures to diversify its lineup by securing new funding for next-generation console projects. Interplay Entertainment has secured more than \$17 million in new funds through a combination of stock placement and credit extension. The company has completed a private placement of common stock to raise approximately \$12.7 million in equity capital and has arranged an additional \$15 million by opening a new working capital line of credit secured by Interplay's company assets and a \$2 million personal guarantee from company chairman Brian Fargo. The new funds will be used to reduce Interplay's outstanding debt and replace a line of credit secured by Titus Interactive, saving the company as much as \$1.5 million in interest payments this year.

Indrema and Digiscents bite the dust. The struggling technology market has claimed two would-be game industry innovators. Indrema had been promoting an Xbox-like game console that sported a 600MHz x86 processor, Nvidia GPU, and 10GB hard drive using a Linux operating system. The company had hoped that the open source nature of the project combined with Indrema's decision not to impose royalties on developers would lure independent game makers to the platform.

Financial realities, including a harsh market climate for Linux-related businesses, forced Indrema to lay off its 50 employees when the company was unable to secure additional funding.

Another unlikely game technology met a similar fate when computer odor purveyor Digiscents likewise failed to secure additional funding. After running through \$20 million in initial financing and distributing 5,000 SDKs, Digiscents was ultimately unable to bring any scent-enabled titles or its scent-enabling peripheral to market.

Low sales cut THQ's revenues and earnings.

THQ reported first-quarter income and revenue well below last year's, but the results were still ahead of analyst expectations. The company's net income fell to \$860,000 on revenues of \$59.3 million in the first quarter, down sharply from income of \$3.9 million on revenues of \$70.4 million in the same period one year ago. Earnings per share fell to four cents from last year's 18 cents per share. THQ cited slow sales and a cyclical lull as the chief causes of its lowered earnings. 🐛



UPCOMING EVENTS CALENDAR

2001 INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE

WASHINGTON STATE CONVENTION & TRADE
CENTER

Seattle, Wash.

August 4-10, 2001

Cost: \$610 (member and student
discounts available)

www.ijcai.org

LINUX WORLD EXPO

MOSCONE CONVENTION CENTER

San Francisco, Calif.

August 26-30, 2001

Cost: \$25-\$1,550 (early bird discounts
available)

www.linuxworldexpo.com



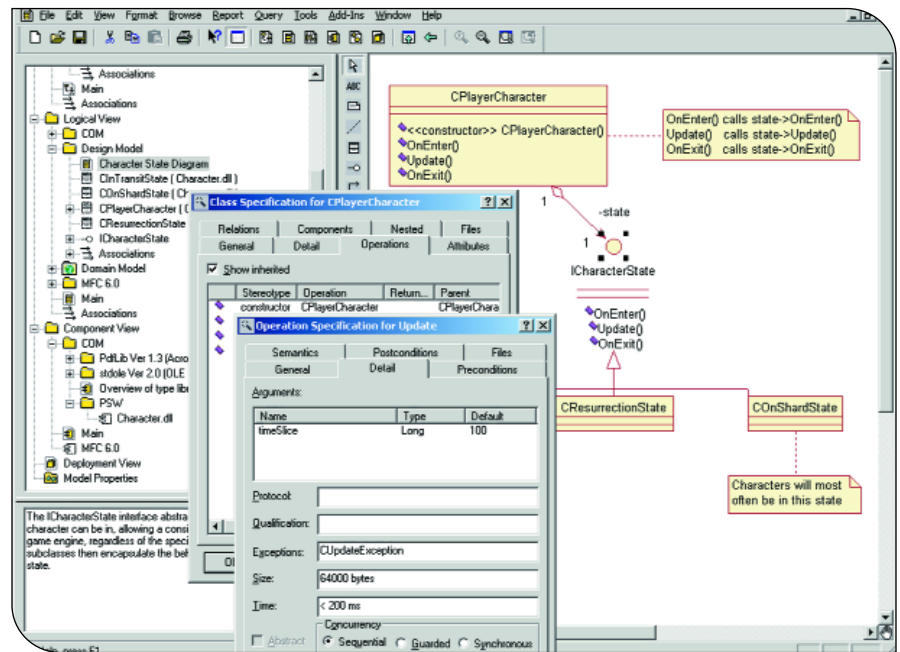
Rational Rose

by jonathan sari

Under the review spotlight today is Rational Rose 2001 Enterprise Edition by Rational Software. The version I'm evaluating is the Windows version, though Rose is also available for Unix. Rational Rose is a tool for creating graphical Unified Modeling Language (UML) models of software. Notable competitors to Rose include TogetherSoft's Together ControlCenter, Popkin Software's System Architect, Telelogic Tau, and, for open source aficionados, Tigris' ArgoUML.

Rose creates and manages software blueprints. The language of these blueprints is UML, the de facto standard for object modeling. Object modeling, a common component of many object-oriented analysis and design methodologies, is the practice of creating diagrams toward better understanding and communicating the complex systems that comprise object-oriented software. As a practicing programmer, you probably do some informal object modeling, be it drawing C++ classes on a whiteboard or simply staring blankly off into space considering how to break up the problem du jour. Rose simply supports, formalizes, and records that process.

Rational Rose models all of the diagrams of the UML: use case requirements, process flows, state machines, software components, physical deployment topology, static class structure, and the dynamics of object interaction. The underlying state it maintains allows the various diagrams to refer to the same objects. Suppose one diagram illustrates how a socket class connects to a player chat class, and another diagram describes how that socket class operates in the multiplayer command pipeline. The model underpinning the whole diagram framework tracks the relations, and the socket class can show all connections from all diagrams (or even connections that don't exist in any diagram). This can come in



The Rational Rose drill down interface for a class diagram.

handy when reimplementing the socket object using UDP instead of TCP, giving a quick and easy road map of the areas of code that will need to be regression-tested. Also, since all of this data lives within a single tool, it offers easy traceability of requirements to actual classes, classes to components, and components to deployment packages. This enables a quick display of which design elements have been implemented, which interfaces live in which components, and where all the components fit in the overall framework.

Rose's user interface uses the familiar browser/MDI document window paradigm. The browser window exhibits the various views on the model, its hierarchy drilling down into packages, diagrams, and classes, while the main window hosts the various diagrams that are the heart of the application. A documentation window shows any documentation for the currently selected element, and a log window logs all messages generated by Rose itself. Model navigation uses the browser window or drill-down within the diagrams themselves, linking the diagrams into a conceptual whole that can tell the complete story of the design. Pop-ups enable access to information not available through the diagrams; toolbars are customizable and dockable. A scripting language that smells like Visual Basic and a

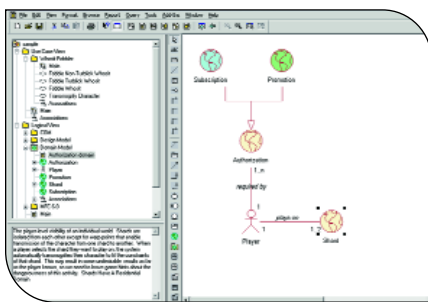
comprehensive COM model allow automation of Rose as well as calls to other applications. Icons used within the model are customizable and vary based on element stereotypes as well as primary element type, while Rose's layout engine automates common-sense display of diagrams. Drag-and-drop makes it a snap to generate new diagrams illustrating new aspects of existing elements: if one class, for example, inherits from or contains an instance of another class and both classes are dragged into a diagram, the diagram automatically adds the relationship.

Rose can forward-engineer stubs in Visual C++ (including ATL COM components), Visual Basic, and Java, as well as generating XML Document Type Definitions. It can also construct models by reverse-engineering code in any of its supported languages, including MIDL and binary COM components via Type Libraries (see Rational's documentation for a complete list). When it generates code — limited to class definitions, instance variables, and method stubs — it inserts tag comments with GUIDs so Rose can alter its interfaces appropriately when prototypes change. It ships with complete prebuilt class models, including helpful diagrams, for MFC 6.0, ATL 3.0, the VB6 standard libraries, and others. Rose integrates with Visual C++ as well as any source code con-



trol systems that conform to the SCC API (most of them, including Visual SourceSafe and Perforce).

If those are the bones, what's the meat? My first and simplest design step when approaching a new system is building a domain model that is a description of the problem domain. When I can clearly and precisely identify the concepts I want my software to model, along with how those concepts relate to each other, I've set the vision for the entire engineering process. Rose lets me do that quickly, cleanly, and with all of the precision of UML, guiding me to illustrate the essential semantics of the system, semantics that can easily get lost in the detail of the problems. But



A high-level domain model.

object modeling is where Rose is worth its weight in gold.

Object modeling is an amazing tool for managing complex systems. With software, the only real limit that defines what we can create within hardware constraints is the level of complexity. The systems enabled by the object-oriented programming paradigm are dramatically more complex than the systems of structured programming. By diagramming code visually and describing aspects of the system in isolation, I can build my own intuitive model of the overall system, encapsulating particular understandings and building the massive complexity piecemeal. Wait, doesn't creating simple diagrams achieve that? Well, I don't know about you, but the system I build never ends up being exactly the system that I had designed. How often do the original design documents represent the actual system as constructed? With Rose I can construct the initial diagrams, generate stubs, and then implement. Through the process of implementation, unanticipated complexities invariably rear their ugly heads. By

bringing the code back into my modeling tool, I can reevaluate the architectural model based upon the real necessities of functional code and keep the model up-to-date in the process.

As a communication tool, UML is invaluable. It'll be nice when I don't have to muddle through a new ambiguous and shifting design language consisting of arbitrary boxes and lines for each new design diagram. Rose delivers, through the UML, a consistent and complete language for representing the essential aspects of object-oriented analysis and design. By using Rose's built-in web publisher, those diagrams can be communicated to anyone with a web browser, displaying a whole system or just describing particular aspects.

All is not wine and roses, however. With any significant tool there's a learning curve, and Rose is no exception. The speed, or lack of same, with which Rose operates is agonizing. Reverse engineering a 10MB Java .JAR file took more than 10 hours of CPU time on a 1GHz CPU; plan to do other things while it is reverse engineering code, loading models, generating diagrams, or web publishing. It's also piggy about memory. Rose balloons to over 280MB of memory with its substantial object models loaded. Additionally, Rose's engineers could learn a trick or two from the game industry regarding user experience. There's no way to cancel out of many of the more time-intensive activities, and the whole UI becomes useless while the tasks are operating. A little work on the threading model would allow me to review other diagrams while reverse engineering is going on. Fortunately, I've never had any of my inadvertently time-consuming activities totally lock up the system on me. Rose will finish its process, if you can wait for it.

The biggest black mark of all is the price tag. Rational Rose lists at \$3,600 for a single node-locked license, plus \$660 to pay for a compulsory year of support. For larger organizations, floating licenses (at \$5,400 per license) allow many installations of the software, with networked licensing software that prevents access when other people are using all of the available licenses. Rose is also available in language-specific versions at a reduced rate if you do all of your engineering in

C++, Java, or Visual Basic.

All in all, Rose is a gigantic tool, both in terms of its memory footprint and its actual functionality. Because it is capable of so much, it's easy to miss functionality, and it isn't possible for me to cover everything it does here. Overall, Rose is the best tool I've found for creating elegant, maintainable, and reusable designs rather than tangled webs of software that mostly function.

A plethora of marketing information, including some valuable whitepapers, can be found on Rational's web site. Two books exist to kick-start the beginner: *Visual Modeling with Rational Rose 2000* and *UML* by Terry Quatrani (Addison-Wesley, 1999), and *Mastering UML with Rational Rose* by Wendy and Michael Boggs (Sybex, 1999). The Quatrani book, in particular, comes highly recommended. 🐉

RATIONAL ROSE ★★★★★

STATS

RATIONAL SOFTWARE

Cupertino, Calif.
(408) 863-9900

www.rational.com

PRICE

\$3,600 for single node-locked license
(plus \$660 for mandatory one-year support fee); \$5,400 for floating license.

SYSTEM REQUIREMENTS

Windows 95/98/2000 or NT 4.0. 64MB of RAM (128MB recommended). 100MB of disk space. SVGA-compatible display, and pointing device with at least two buttons.

PROS

1. The most usable modeling tool.
2. Provides reverse and forward engineering of most important languages.
3. Integrates with existing source control and development tools rather than trying to replace them.

CONS

1. Steep learning curve.
2. Poor user interface handling during CPU-intensive activities.
3. Slow and piggy.



NEWTEK'S LIGHTWAVE 6.5

by david stripinis

If I had only one word to describe Newtek's Lightwave 6.5 it would be "powerful" — but "quirky" would be a close runner-up. Lightwave 6.5 is the latest release of the venerable 3D modeling and animation program that offers powerful tools to create pretty much any 3D asset your game could need.

The interface is, to be generous, unique. Lightwave is actually two separate programs, Layout and Modeler. Layout is where everything from animation to lighting and rendering is accomplished, while Modeler is dedicated to, and this should be obvious, modeling and texturing. While this may seem odd if you're used to all-inclusive programs like Maya or 3DS Max, I found switching between them no different from switching between menu sets or panels in other programs, and it actually offered an advantage I will discuss later.

Modeler is an extremely robust and versatile tool for creating polygonal models. Both organic and mechanical forms can be created with equal ease, which is quite a bit easier than in many of Lightwave's competitors, I might add. Modeler seems to follow the theory that less is more. Rather than an overwhelming suite of tools which serve more to confuse rather than aid in art creation, Lightwave encourages you to use the same tools over and over. While a large variety of tools are offered, I found that using the bevel, translate, and stretch tools in conjunction with Lightwave's truly amazing symmetry feature allowed me to create pretty much anything I wanted with ease. The recent addition of real UV coordinates drastically improves the texturing capabilities of Lightwave over previous versions.

No discussion of Lightwave's modeling capabilities would be complete without a discussion of its subpatch objects. While it's true that subpatches are simply an implementation of subdivision surfaces, Lightwave has helped to popularize this method of working and provides a nice workflow for working on complex organic forms.

Modeler is also where users build their "skelegons" — Newtek's new skeletal deformation tool. By incorporating these

skelegons into Modeler, all your modeling tools are available for modifying your character's skeleton. This allows the user to build a working character quickly without having to learn a completely new toolset for skeleton construction. Also new is the addition of weight maps, which allow for advanced weighting of surface points to bones. Interactivity in Modeler is very high, and the viewport display is quite intuitive.

Animation itself is also quite nice. All animation is accomplished in Layout. Lightwave's IK is fairly intuitive, and its solver gives results pretty much as you'd expect from any professional package. Animation doesn't really need a lot of tools. It is pretty much a case of setting keyframes and editing function curves. The graph editor has many editing tools, though most animators simply need to adjust curve handles and move keyframes around.

One truly neat benefit of Lightwave's separate programs is being able to go back and forth between a model in its rest pose and the actual animation of that model. If you notice you need more detail in the shoulders to prevent crumpling during a bone rotation, or you need to completely change the design of a character, you need only change the model itself. This interaction between Modeler and Layout, while not exactly an exciting item for a marketing brochure, is really a nice feature appreciated by artists actually working in production.

Lightwave 6.5 also incorporates a few improvements to its world famous renderer — including caustics, global illumination, and 128-bit rendering — which gives fast, clean, film-like results. The viewports also offer many WYSIWYG rendering features such as lens flares and fog. Lightwave 6.5 also incorporates VIPER, an interactive renderer, the latest trend in 3D packages.

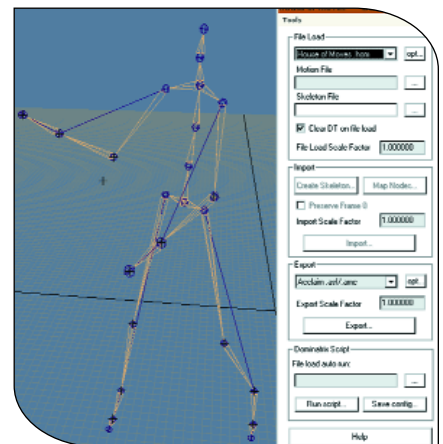
Lightwave 6.5 offers any artist good animation tools, great rendering technology, and unbelievable modeling tools, especially for creating characters. Affordably priced (\$2,495) and available for a wide variety of platforms, Lightwave offers a way to round out the weaknesses of whatever toolset you may now use.

★★★★★ | LIGHTWAVE 6.5
Newtek | www.newtek.com

HOUSE OF MOVES' DOMINATRIX FOR MAYA

by christopher clay

The first time I tried out Dominatrix, I had received a zip file containing the software and some sample motion capture data. My task was to apply the data to a nine-foot-tall hulking giant. After a quick install-and-unlock process for the 15-day free trial period, I set to the task. Roughly 15 minutes after first opening Dominatrix, my character was dancing like the svelte motion capture actress the data was based



Motion capture data in Dominatrix.

on. As wrong as that looked, it left a lasting impression on me; I had never worked with motion capture data before, and the whole process had worked almost seamlessly.

Dominatrix should be considered by developers who need to apply motion capture data to a number of different characters with varied scaling, or who want to apply, retarget, and edit mocap data in-house (for example if you know your game's characters are likely to change over the course of the production), and who want to keep this kind of data manipulation in Maya. It's also an attractive option for those looking for an easy way to loop most of their motion capture data. From this standpoint, the biggest deterrent is the price tag: Dominatrix runs \$5,000 for one license and \$2,500 for each additional seat.

At its most basic level, Dominatrix is a very simple plug-in for Maya 2.5 and 3.0 that even the most nontechnical artist can use to perform basic motion capture application tasks. I can't stress enough how sim-

- ★★★★★ excellent
- ★★★★☆ very good
- ★★★☆☆ average
- ★★☆☆☆ disappointing
- ★☆☆☆☆ don't bother

ple it is to apply basic motion capture data with Dominatrix. The configuration-saving feature makes it easy to simplify the entire process for nontechnical artists. For those who are technically savvy, it has excellent scripting and batch properties which allow



Robots created with Dominatrix.

the user powerful access to the tool. Among other things, these allow for more fine-tuning when re-targeting motion capture data to a differently sized skeleton.

An impressive ability of Dominatrix, which

helps greatly when looping motions, is the ability to convert traditional FK motion capture data to IK data. Though setting this up in Maya and within Dominatrix is technical and time-consuming, it is very effective at maintaining data quality when remapping data to nonstandard skeletons. I also found it much easier to loop motion capture using the IK solution. It should be noted that House of Moves provided excellent technical support while I was learning this process which greatly smoothed out the stumbling blocks I ran into.

Another handy feature is the ability to export animation out of Maya to motion capture files. This allows users to export traditionally animated moves to motion capture files and re-target them to different game characters, providing them with a great base motion that an animator can then quickly recharacterize.

If your production is in need of a quick solution to importing motion capture data, this is an excellent choice for you. Though the cost may seem steep at first, when you consider the cost of having your in-house engineering team write a similar plug-in, it becomes negligible. Dominatrix is an easy tool for beginners, with enough depth to satisfy those who desire greater levels of control. If you're working on motion capture in Maya, let Dominatrix whip your production into shape.

★★★★★ | DOMINATRIX
House of Moves | www.moveshack.com

BEGINNING DIRECT3D GAME PROGRAMMING

BY WOLFGANG F. ENGEL AND AMIR GEVA

reviewed by mark deloura

I had high hopes for this title. I've been using OpenGL and its predecessors for about 10 years, and since any 3D programmer worth his or her salt these days needs to have a working knowledge of both OpenGL and Direct3D, I decided to read through this book from Prima Publishing. It seemed like a great way to quickly come up to speed on Direct3D. Boy, was I wrong.

Beginning Direct3D Game Programming is about 500 pages, and comes with a CD that includes source code, the DirectX 8 SDK, and trial versions of a few programs and games. The first half of the book is devoted to introducing you to Direct3D 8; it then moves on to cover how to use QUAKE 3 .MD3 files, and follows with introductions to C++, Windows programming, physics, collision detection, mathematics, and the Common files that are a part of Direct3D 8.

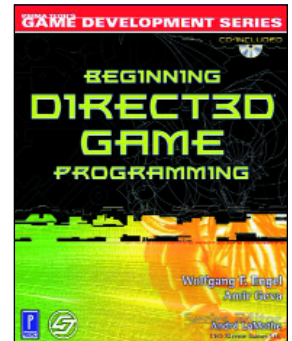
One great thing about this book is that it covers Direct3D 8 and provides a bit of history by discussing some of the changes that have been made through the various versions of Direct3D. Unfortunately, it glaringly omits what are likely the most revolutionary features of Direct3D 8: vertex shaders and pixel shaders. While the book is designed for beginners, it does go into multi-texture and multi-pass techniques as well as complicated subjects such as anisotropic reflection. So the vertex and pixel shaders aren't left out because they're too complicated. This omission definitely makes the book less valuable.

But unfortunately this book won't be very valuable for you anyway. From the very first page of text, it is littered with errors. Some of the errors are grammatical, many are technical, and there are numerous formatting errors, all of which add up to a book that was obviously put together in a hurry. The whole thing is almost unreadable as a result, because you don't know which parts to trust. As an example, for a book that targets beginners, leaving out the exponents on nearly every equation in the entire book is completely unacceptable. I'm serious. For example, take page 475, which has 42 missing exponents in

statements like this one:

$$2 * \underline{U} * \underline{V} = \underline{U} + \underline{V} - (\underline{U} - \underline{V})$$

Other errors are not as dramatic, but are still highly confusing. The printed code is formatted entirely haphazardly, wrapping over page boundaries and pseudo-aligned in some random fashion (this works particularly poorly in the case of printed matrices). The description of 3D rotation is just plain wrong. Some of the images are clearly incorrect. (The "right" vector goes out the back of my head?) Sometimes the images and text are both incorrect. (Setting the texture coordinate range to [-0.5, 1.5] makes the texture repeat 1.5 times?) The text in the first half of the book is frequently confusing and occasionally incorrect, though fortunately the readability improves after about 150 pages.



I'm frankly not certain what the point of some of the accompanying chapters is. They're not bad, just not necessary. Someone who is doing Direct3D programming probably doesn't need an introduction to C++. Also, strangely enough, the entire physics chapter is just eight pages, with the first two pages concentrating on 3D math. Fortunately, the collision detection section is a hefty 40 pages, and the Windows, C++, and math sections are similarly well sized.

The CD which accompanies the book is useful. Since you can't trust the accuracy of the material in the book, you can copy the code from the CD, compile it, and test things out for yourself. The programs on the CD correspond to the chapters of the book. They are well designed and easy to tweak.

The frustrating thing about this book is that it could have been a great introduction to Direct3D and game programming for beginners, as well as being useful to 3D veterans as a Direct3D introduction. Unfortunately, it is neither.

★ | *Beginning Direct3D Game Programming*
Prima Publishing
www.primapublishing.com

Don Daglow

Breaking Typecasts

Don Daglow began his career in 1971 with a baseball game written for a PDP-10 mainframe. He joined Mattel as one of the first five Intellivision programmers in 1980, worked as a producer at Electronic Arts, and later headed up Broderbund's Education and Entertainment division. Over the past 30 years he's had a hand in more than 100 games, including the first mainframe RPG in 1976, the first sim game (1982's *UTOPIA*), and *NEVERWINTER NIGHTS* for AOL (1990). He's been hanging his hat in the big cheese's office at Stormfront Studios since founding the company in 1988.

GD. You've developed for everything from mainframes to Intellivision to Xbox. There aren't very many people who have been around that long who are still actively developing games. Why do you think that is?

Don Daglow. I think first of all you have to be old [laughs]. And a lot of people who are in this business were born at the wrong time so they can't yet be old. It's a mathematical advantage. Kidding aside, I think part of it is the paradigm shift in the industry. For a lot of us who were in the one-person-in-their-spare-bedroom era and were comfortable in that era, the idea of teams of people and shared and collaborative responsibility I think was uncomfortable.

GD. Did you have to learn successful project management piecemeal over the years, or did you find a few fundamentals early on and iterate on them for bigger teams and projects?

DD. The half that for whatever reason I got pretty much right at the start was the cultural part, because I really believed that having the right culture was a key part of the process, and ironically that seems different from project management. But there's no better way to ruin a great plan than to have the culture be awry and have people be miserable. Being at EA in the early 1980s, I got a chance to see that idea of "build a great culture and it will help you build great games" put into action. The part that I learned bloody piece by bloody piece by making mistakes is what formal project managers in aerospace learn from day one, which is the mechanics of how you go about doing it. For example, there's a very standard process of reviewing an initial project plan from the ground up to make sure the budget, features, and schedule all match. I've only learned that part by scraping my knees and my elbows. I still feel like I have things to learn in that category.

GD. How has the shift to next-generation consoles gone over with your team? Are you leaning toward continued multi-platform development or might you home in on one platform you consider the strongest?

DD. Never, ever one platform — I learned that on Intellivision! It's all about a few key things: Diversity of clients, so we have a two-franchise limit for any one client; diversity of platforms, to split the risk; and diversity of genres — we periodically get typecast for things. We work very hard to break that typecast. In the early 1990s it was RPGs and baseball. Then it was team sports in the mid-1990s. In the late 1990s it was motor sports. Now

because we have so much RPG and adventure going on I think we're going to go back to being typecast for that, and we're already taking projects to try to break that typecast.

GD. How important do you think it is for development studios to remain independent?

DD. It's worked well for us. The metaphor I use is that when you meet the right partner, in terms of true love in your personal life, you know it. You can't orchestrate it and say, I know it will happen at this moment. So you can't say, I'm going to go out this quarter, or this fiscal year, and I'm going to meet my perfect partner. On the other hand, if do you meet that perfect person, you have to surrender to them. The reason we're still independent is because we have not yet had a situation come up where that perfect partner came along. There's no such thing as an idea that we are by nature independent. We're simply still single.

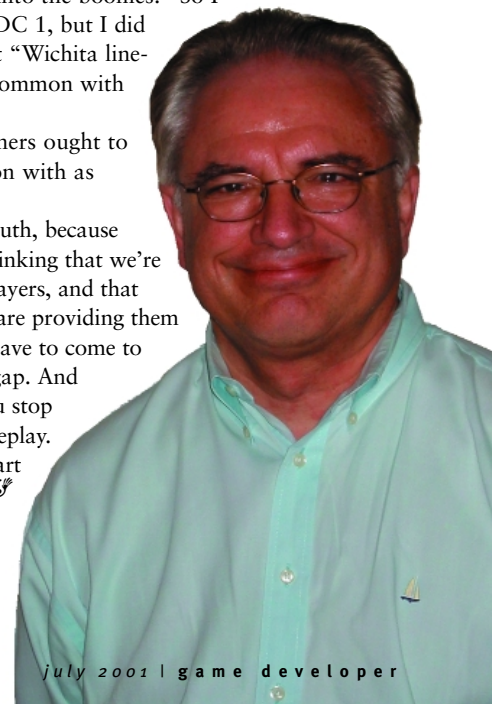
GD. Have you learned any memorable lessons about game design?

DD. I stopped taking design credits about 10 years ago. But on my way to the first Game Developers Conference, which was at Chris Crawford's house in the hills above San Jose, I was going up this narrow road, and there's this construction worker who says, "Road's closed for two hours, detour this way." So I take this detour, and I come up on this other detour that puts you on this gravel road. Then there's another detour sign onto what is basically a fire road. So I go down this and I'm thinking, this could not possibly be right — I'm under electrical towers on an access road. I come back down to retrace my steps to figure out where I went wrong, and there's a construction crew there, and they have this look of sheer delight on their face. And I realized that I was in fact a player in their own game, which was "take the passerby and divert them into the boonies." So I ended up being late for GDC 1, but I did make it. It just proves that "Wichita line-men" have something in common with game designers.

GD. Perhaps game designers ought to have something in common with as many people as possible.

DD. That's an essential truth, because the minute that we start thinking that we're somehow separate from players, and that we in our infinite wisdom are providing them with something that they have to come to us for, we start to build a gap. And once that gap happens, you stop thinking about selling gameplay. I've seen a lot of careers start to unwind at that point. 🍷

RIGHT. Stormfront's Don Daglow.



Cleaning Up the Garage

Usually I dive right into my monthly topic without letting my wandering thoughts carry me too far away. Last month I left off discussing how to create an artistic background using simple fluid dynamics. I will get back to that in a minute, but

I wanted to discuss some general game development issues that have been on my mind lately.

My game development career actually began with a high school track meet. My friend from school, who drew comic book art on his PeeChee and fancied himself quite an artist, started a little side business. We both had Apple II computers and were already hooked on games. We started looking for a way to fund our toy habits. Our first “product” was a simple database program for the track coach to keep stats at the meet. We used that money to buy a copy of one of the latest games out at the time, WIZARDRY. That game was amazing. I played it for hours on end. When I got frustrated that I couldn’t find the best equipment, in the true hacker spirit I started editing the save game files to goose my stats. When friends started to ask for copies of the program, we bought some blank floppy disks and started selling disks with our little programs on them. We created a slick editor for ULTIMA that let you edit the character and object graphics. (This was all 20 years ago!) We packaged up some of the wacky games we had created and put them on a disk and sold them at the local computer store. We started designing our epic adventure game on a budget of \$40. We plot-



JEFF LANDER | *When not playing in the garage planning his latest epic game project, Jeff can be found hard at work at Darwin 3D. Send tales of your garage game development experiences to jeffl@darwin3d.com.*

ted out all the graphics on graph paper and tacked the concept drawings all over the room. We finished about five rooms of the adventure before we decided to blow the remainder of the budget on plywood and build a skateboard ramp in my friend's backyard. Another project was the victim of irresponsible management.

In those early days, just like now, many people were developing games. All you needed was a little bit of technical skill, even less artistic skill, and a few dollars for supplies like blank disks and zip-lock bags. The history of the early Apple II days is filled with tales of small groups of people putting together games in their spare time that have endured and are even copied today.

I get asked a lot whether I think that the days of garage game development are over. It is an interesting question that I have thought about quite a bit. Even back in those "good old days" game development was a hard business. The game-buying public was much smaller. Commercial success still relied largely on getting publishing deals to ensure your game was available at the hundreds of little computer stores around the country. Most game projects probably never were completed and most that were finished probably lost money. However, clearly a lot of things have changed.

The public is now not only game savvy, they devour the stuff at an incredible rate. Games go from being hot new releases to anchors at the bottom of the bargain bin in a couple of months. Access to those bargain bin shelves is not controlled by small enthusiastic shop owners. It is controlled by national game buyers looking for a quick resale. Name recognition, flash, hype, and sure-thing are the rules that govern modern game distribution.

Flash and hype are expensive to achieve. Game graphics that can compete for attention these days need to be at or very near movie quality. Creating high-quality animation requires very skilled artists and technicians. Content deep enough to impress the game press takes a great amount of time and effort to create.

Most developers will quote that a full game project takes about 18 months to complete. In practice, I have rarely seen projects (including simple sequels) hit that timeframe. In addition to the time required to create the actual game, the amount of money needed is staggering to a small garage development group. PC game budgets are averaging in the millions of dollars. Console development is even more difficult. While high-end console games certainly have a better chance of recouping the development money spent, the market is not open. Since the console publishers will generally spend much more than the development budget on marketing the product, they want to be pretty sure they will make the money back. You need approval just to get started on the game platform.

Is It Possible?

Are garage games possible in this climate? It is certainly possible to create a game with a small group of developers working on an extremely limited budget. Programmers and artists of all kinds have gotten together and created impressive games and



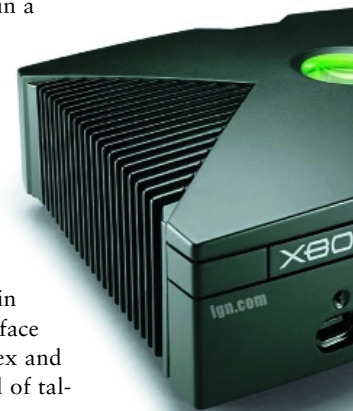
WIZARDRY (left) and ULTIMA (right), two of the first computer role-playing games.

game demos. I am impressed every year at the Independent Games Festival by the quality of the games that people are able to create. Many of these are at or near commercial quality. However, does this mean that these small groups are actually making money? It is hard to imagine that they are.

The Internet has provided a marketplace of unprecedented size. Now independent game developers are not limited by the number of zip-lock bags they have or the number of computer stores within bicycling distance. The entire world is available for marketing, sales, and distribution. Many people discussed the idea that increasing access to the market via the Internet would be the end of the current distribution and sales systems for many industries, including music and games. That may eventually happen, but it is not going to happen anytime soon.

The Internet audience is pretty jaded. There is a great deal of products out on the Internet free for the taking. While thousands of people might download a developer's latest game demo, very few of those downloads would potentially turn into actual sales. I have not yet heard of a game that made any significant money through Internet sales. If not backed up by a well-funded and organized advertising campaign or a partnership with a commercial organization with a large audience such as AOL or RealNetworks, real success would be difficult.

Since getting money directly from consumers via the Internet is so difficult, advertising models have dominated the Internet game development scene. Certainly developers of web games for systems such as Flash and Shockwave have had quite a bit of success. Web site owners seem willing, at least for now, to pay developers to create simple little games to draw viewers to their web sites. Since the capabilities of the systems are pretty limited, a small team is able to create a project in a reasonable amount of time and charge fees that web site operators are willing to pay. However, creating a game of any significant scope is going to require talent and time that may break the back of this funding model. Certainly the emergence of 3D technology in these online systems has started to signal this. As the technology and content bar gets raised in the online space, the same hurdles that face developers of PC games appear. Complex and sophisticated games require a great deal of tal-





CHASE ACE 2 (left) and HARDWOOD SPADES (right), two of this year's Independent Games Festival finalists.

ent and time to create. Licensable game engines can help with this to some extent. However, as anyone who has worked on a licensed game project knows, a game engine doesn't necessarily make the process of creating a game much easier.

Other ideas for Internet distribution such as episodic development and delivery hold some promise. However, until it is proven that players will actually pay for this type of product, it would be risky to count on it to fund your game project.

If business on the PC and Internet side of things seems a bit bleak for the garage game developer, the console business is downright depressing. Modern console games for systems like the Playstation 2, Xbox, and Gamecube require extremely advanced game engines simply to compete for the publisher's attention. Developers need to be able to create an entire CD, or even worse a DVD, filled with content. To actually get a console product to the state where it could be published, a garage developer would have to have a great many talented friends willing to work a long time on a small chance. That time is probably much better spent creating a demo and approaching publishers for development money in a typical publishing scenario.

The Xbox is held as a great opportunity for garage developers. Unlike the other high-end consoles, developers can get started on the same tools they use for PC work. The technology and specifications are largely familiar to anyone who has worked on PC 3D hardware. However, this does not lower the content bar. Microsoft will most likely not allow any game to be released for the Xbox as it is for the PC. In order to be released to the market, your game will need to be approved. That means it will need to be of high enough quality to compete with the other Xbox titles on the market. Again this comes down to talent, time, and development dollars, all things the garage developer does not likely have.

Stop Fiddling in That Garage and Get Some Work Done

So what can a small team of independent developers really hope to accomplish? If you really want to get a commercial game out on the market there are only a few options. You can take a job with a

game company that has the talent and funding needed to complete a project. You could also self-fund a demo and design that is good enough to approach publishers with for development money. However, you are going to need to be able to convince them that you are capable of building a team large enough to create the project in a reasonable amount of time. No publisher is going to risk the development money for a cutting-edge title on a small, inexperienced garage team. They are going to want a large team in an office with the capacity to get the project done on time and on budget.

The handheld game market is particularly interesting to me these days. While the Game Boy never really excited me very much, it probably should have. Here is a system with millions of avid game players. A small garage team can easily handle the graphics and coding needed for a game project. Getting a project approved still requires jumping some obstacles, but the commitment is not nearly as scary as the high-end consoles. The emergence of cell phone and PDA games also offers opportunities for garage developers.

The new Game Boy Advance is even more interesting. The system is still largely 2D sprite based and there is no 3D hardware. However, it looks like a pretty sophisticated little computer. This system may provide a great opportunity for garage developers to create some exciting projects.

The truth is that success as an independent garage developer requires a bit of compromise. The developers I know who are making it work are not spending most of their time creating their own projects. They feed their garage project habit with a diet of contract work, consulting, and depletion of savings. This is obviously not a recipe for quick game development. Ask any independent developer how many pet projects they have on the shelf at any one time and you will see what I mean.

Overall I think it is a very interesting time for independent garage game developers. While the high end will likely be dominated by traditional development and publishing methods, opportunities exist for the small group. Experienced developers who are burned out and jaded by the development grind can band together to form exciting new opportunities.

So what are you waiting for? Take up the challenge. Grab some of your most talented friends and start cracking. You have a lot of work to do before you can start proving that it can be done. Let me know how it is going.

Back to Business

Time to shelve my pet projects and get back to work. Last month, I was discussing how I could use fluid simulation techniques to create organic and artistic images ("The Era of Post-Photorealism," June 2001). I created a complex fluid field using basic flow elements; now it is time to actually do something with the flow field. The basic flow elements combine to form a velocity field. Using the flow formula from last month's column, I can quickly determine the direction vector at any position in the flow.

The easiest way to create an image from the flow field is to drop random particles of different colors into the flow field and let them simulate through the flow. To integrate the position of



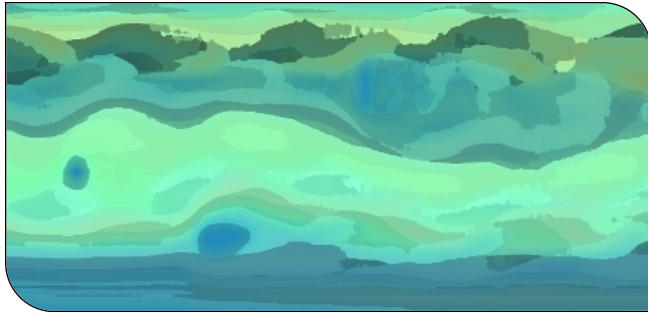
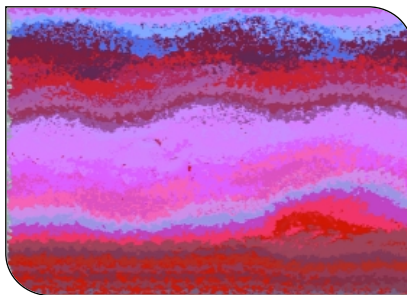


FIGURE 1 (top). *Liquid Sunset.*

FIGURE 2 (right). *Fiery Morning.*



the particles forward through the velocity field, I am using the predictor-corrector integrator I first described in my column last year (“Pump Up the Volume: 3D Objects That Don’t Deflate,” December 2000). Since there are no constraints on the particles other than the flow field forces, this integrator is more than robust enough.

By varying the particle size, I can create a sort of pointillistic view of the flow. This alone is not really very interesting to watch. A much more compelling image can be created by drawing the particles as a connected line of color. Since the particles are governed by the fluid flow, these connected lines will be very smooth curves. A variety of smooth strokes can be created by controlling the number of points in the connected line and varying the frequency of the particles along the line. Simply drawing the line by connecting the dots is not really very interesting unless you are trying to achieve a look like a moving etching. However, the lines can be given width by creating polygons that have vertices offset from the stroke line. This width can be randomly varied along the length of stroke much as I described in my column on cartoon-style ink strokes (“Deep in the Programmer’s Cave,” May 2001). Those stroke polygons can be rendered using a solid color or with texture. I found that an interesting watercolor style of display can be created by rendering each stroke with a texture that is mostly translucent using the alpha channel. These strokes are then layered and blended together to make the final image. You can see the results of one of these experiments in *Liquid Sunset* (see Figure 1). Since the system is based on fluid flow, when this watercolor image animates, it does so in a very fluid manner.

A different and more turbulent effect can be created by changing the rendering technique using the same fluid flow. Instead of creating long, smooth strokes this time, each particle is rendered as a single billboard sprite using a series of fractal textures that animate in a manner like a cloud evolves. Each

particle in the fluid field is given a life cycle. The particle is born, lives for a while, and then dies. The size of the billboard sprite grows after birth to a maximum size and then shrinks as it dies out. When many of these particles are placed in the field and rendered with these overlapping billboards, the effect is not unlike moving smoke or fire. You can see the results of this in Figure 2.

By simply experimenting with the fluid parameters and the rendering system, a great deal of effects can be achieved. These systems can be rendering in an off-screen buffer and then used as texture maps for an environment. Given the performance of current 3D graphics hardware, it is quite possible to use this technique to generate particles and polygons in the game and then render them as normal 3D objects that can be applied anywhere.

I experimented a lot with animating the parameters of the flow field. But by changing the position and setting of the basic flow elements dynamically, even more complex effects can be achieved.


Extending to 3D Objects

This method works well for creating backgrounds and complex textures for flat surfaces. However, the ideas of potential fluid flow are useful for 3D systems as well. I am probably not going to be able to create a 3D grid of flow elements and fill the room with 3D particles. But, in limited application, some complex effects can be created.

The potential flow formulas that I described last month can be easily extended to 3D. Once that is done, I can calculate the velocity field at any point in a 3D space. A 3D particle dropped into this flow field could then animate through the field just as it did in 2D. This is really just an extension to the basic idea of 3D particle systems. In a particle system, particles are dropped into a simulation space and various forces act upon them as they move through the world. In addition to the normal forces used in particle systems such as gravity and wind (which are really just types of uniform flow using last month’s definition) I can add the other flow elements such as vortex and doublet flows. High-end 3D applications have elements like this built into their particle systems.

With these vortex elements, the particle system takes on a more turbulent appearance. Things like wispy smoke look more realistic when they swirl around a bit. Doublet and vortex elements can be attached to an object that moves through the particle field disturbing the smoke in a realistic manner.

The idea of strings of particles as strokes is also useful in the 3D environment. I can take a simple 3D model of a tree and emit short strings of particles from the faces of the tree. These particles can be influenced by a series of vortex and uniform flow elements attached to the tree. What would that look like? Would it resemble a hairy tree blowing in a swirling wind or would it look like the kind of twisted and expressive trees that Vincent van Gogh created in his paintings? Why don’t you give it a try and see what you can find?

Sample code and demonstrations of the techniques discussed in this and last month’s columns can be found on the *Game Developer* web site at www.gdmag.com. 



Where's the Design in Level Design?

Part 2

Last month I shared some of the benefits of using proven design principles in level design, an array of which can be borrowed from real-world architecture and interior design. This month, I will present even more and relate some of them to actual game levels and to the creation of custom-designed level textures. Just to be clear, this isn't a "how-to" tutorial on designing, modeling, or texturing a game level. Instead it is a collection of considerations to help you with a more efficient execution of good design for your level modeling and texture work. As promised last month, I will also walk you through the steps of prequalifying level assets so that you can avoid making costly mistakes, thus saving you time and money better spent elsewhere.

High Expectations

We have an interesting challenge in the game industry as computer hardware technology plows forward faster than ever. This increase in power has a direct correlation with the player's growing expectation for stunning visuals. As game artists and level designers, we should try not to get overwhelmed or intimidated by the rising technology bar. We have a role to play in perpetuating its progress and should therefore embrace it, at least to the point where it enhances our process of improving and implementing good design that produces believable and engaging levels. This in turn improves the product and the gaming experience. We are further challenged to meet these high expectations with equally sound design principles and concepts. Improved hardware will



TITO PAGÁN | *Tito has been creating art in the game biz for about a decade. Visit his web site at www.titopagan.com.*

eventually make it possible to create game environments that are intricate, highly detailed, and free from technological limitations in performance. Where will you be and what will you be doing when this happens?

So What Is Good Design?

Everyone has his or her own idea about what good design is.

One thing I think we can all agree on is that design is a perceivable and desirable quality that surrounds us in our everyday life, yet we often overlook its importance. It provides comfort, draws our attention, and gives us the visual cues we've learned to depend on for information such as directional and level changes, defining means of egress from within a building, and so on. In general terms, design is the skillful planning and fashioning of the form or structure of an object, a space, a work of art, a decorative scheme, and yes, a game level.

In creating a comfortable and logical game level, a job well done does not leave your player feeling uneasy about the personality, balance, proportions, lines, or character of the space or structure being portrayed. All environments possess these traits. Keep in mind that there are many different kinds of spatial designs that are well suited for a 3D world. Your level design should be one that addresses your individual game's requirements and applies basic design principles.

Some guidelines that govern good design used by other practicing design professionals include balance, scale, proportion, unity, emphasis, rhythm, and harmony. All designs consist of color, pattern, texture, and style, and if these guidelines are adhered to, the player will feel comfortable in an environment.

Balance. Balance is the feeling of equilibrium. How do you feel when your life is out of balance? That is also how a player will

feel when a decorative wall, room, or outdoor space is out of balance. All balance is based on vertical and horizontal axes. Getting equal weight on each side of an axis makes a space in or out of balance. A good analogy would be riding a bicycle or standing on your head.

Scale. Scale is the size of an item in comparison to its surroundings. A piece of furniture or an accessory can be too big or too small for a room, a wall, or a setting. A carpet texture scaled too big creates a "dollhouse" effect in a game. The casual observer is uncomfortable when this occurs. It's just not pleasing to the eye

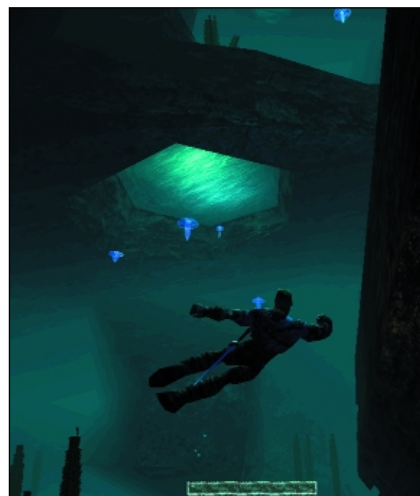
when things are out of scale. A typical example of bad scale is the smaller-scaled furniture and accessories in a game level created with a level editor that uses constructive solid geometry (CSG) brushes bound to a

grid (such as Worldcraft). A popular example is Valve's HALF-LIFE levels. These were created using Worldcraft 2.0, which lacked fine control in the modeling process back then and probably forced the level artists to create and accept badly scaled and disproportional furniture and accessories. The game is still fun to play (one of my all-time

favorites) but can be quite the eyesore in some areas. I discovered these limitations myself when I created levels for Sierra Studios' S.W.A.T. 3 using this editor (see Figure 1).

Proportion. Proportion is the size of things compared to themselves. In furniture, legs can be too large or too small for cushions. Doors and windows can be too long or too big for the walls they are built into. Proportions that are pleasing to the eye will promote feelings of satisfaction. Items that are out of proportion actually create anxiety, and we usually don't want to look at them.

Unity. Unity is the element that carries the theme and scheme of the room. The point of unity within a room can be a paint-



CLOCKWISE FROM TOP LEFT: FIGURE 1. Realistic proportions in S.W.A.T. 3 by Sierra Studios. FIGURE 2. A strong common element of structural support carries the theme in this room. FIGURE 3. The focal point of light ahead directs the player and gives reference. FIGURE 4. Good repetition of likeness creates rhythm and movement.

ing, an area rug, a major piece of upholstery, or an architectural feature (see Figure 2). When a certain style and color scheme are contained in a single decorative element in a room it makes it easy for the player to understand what is going on.

Emphasis. A point of emphasis or focal point is that item or place that catches your visual attention upon first glance. The focal point of the room is essential to anchor the composition of the room. Each time you enter a new space you have an opportunity to create a new focal point. In a composition, the player's eye travels from the focal point to the rest of the room and back to the focal point (see Figure 3). Without a focal point, the eye tends to wander aimlessly throughout the space, searching for something to focus on. This lack of grounding produces anxiety. If the focal point can also be the point of unity for the space, you have accomplished two things at the same time. You have secured the player's attention and unified the room.

Rhythm. A repetition of like items in a room or space that move the eye from one area to another, rhythm can be accomplished with color, pattern, texture, lighting, and style or character (see Figure 4). Think in terms of music. How important are the drums and the bass in a song? Once you have the rhythm of the beat, you are into the music. That consistent beat carries you throughout the music.

In a game level, a certain repetitive motif, pattern, or texture could help guide the player through the experience of movement. Moving or nudging the player through the exploration experience is one of the most basic yet important responsibilities that a level design has to satisfy. When a room or space has rhythm, people feel secure because of the comfort in the predictable nature of their surroundings.

Harmony. Harmony is when a common element exists that binds all parts together. Like a common denominator, this element can be a color, pattern, texture, detail, or the character in a room. In a picture grouping, for example, it can be the frame, matte, accent color, or subject. When the principles of design are adhered to, the result is harmony. All of the parts relate to each other in a way that allows blending and bonding. Harmony is the difference between a great-looking and -feeling place and a room or space full of things.

Texture Design Also Matters

These same basic principles can be applied to the careful design of textures. How many times have you sat in on a design meeting where someone criticizing a level idea has made the following statement: "From the point of view of the player or rate of travel through a level, no one cares about that much attention to detail in the texture"? If you have invested time in the game industry, then you more than likely have heard this several times.

Consequently, your environment's overall look and feel has probably taken a hit.

It is a misconception in our industry to think that as developers who play games we know instinctively how much detail is enough when creating a great-looking game that still runs well. People are diverse, they play games differently, and it is safe to say that perception of what makes a game great will vary from person to person. Giving attention to detail in all aspects of a game, including textures, should be considered crucial. Texture detail and design can't be an afterthought if you are trying to achieve a cohesive look in your levels. And when it comes to detail work, it's the little things we take for granted that count. The rivets, dents, rust, stains, and scratches all give life and personality to a surface (see

Figure 5). If the level designer is not a skillful texture artist, then it makes perfect sense to hear him or her play down the importance of a well-crafted texture set.

An exceptional texture artist is worth his or her weight in gold. This is often expressed by the seasoned art director whose job it is to manage and direct creative resources. This art director also knows that the texture artist has the ability to promote the perception of quality in a product while addressing known issues and constraints one has to consider when creating a texture set for a level. The 2D artist needs to be fully aware of the latest effects supported by current and future graphics cards and help devise creative ways to exploit them, such as real-time reflection and bump maps. Besides helping to establish the final look and mood of a level, the textures also provide the player with important information such as direction, interactive clues, and orientation. If your development budget does not afford you a skillful and dedicated texture artist, then your level designer or modeler has some ramping up to do.

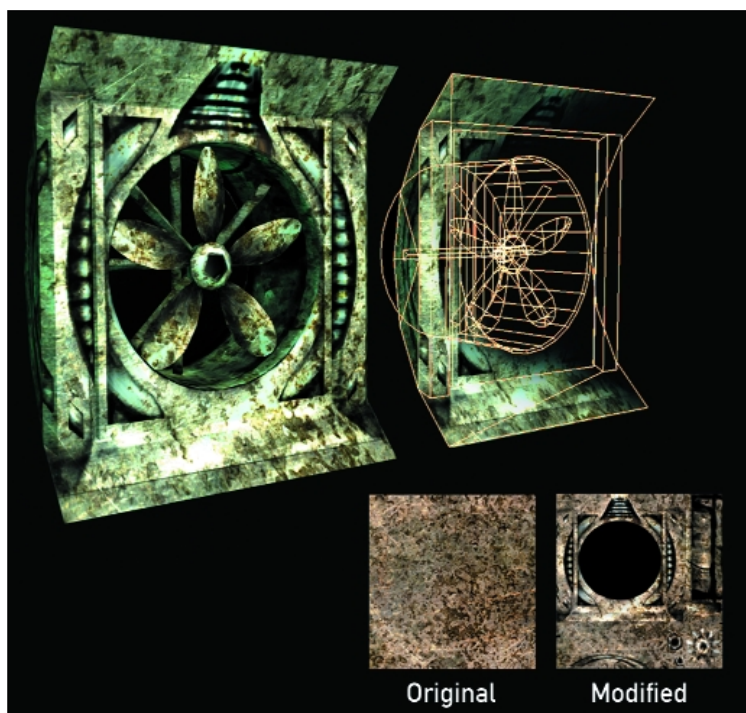


FIGURE 5. Add life and personality to your objects with well-designed textures. This 3D object was created for WildTangent's BETTY BAD.

Make No Mistakes

To avoid making costly 3D art assets that aren't needed or that just don't work, level designers should analyze their needs to determine what is required before creating objects such as furniture and architectural details. Here is a basic guideline you can use to accomplish your goal of creating an attractive 3D interior space to play in and not make costly mistakes in the process. If you are going to borrow from real-world environments for inspiration and creation, you should consider using the design tools other professionals use to create the spaces we live in every day.

If you know what you need, what you want, and you understand basic design, then your chances of making a mistake are almost entirely eliminated. You can start to build with confidence and clear direction. That is, of course, until the game designer changes the general purpose or focus of the game level or space — at which time you simply smile and reapply these basic steps.

Avoiding a Fall

Take yourself step-by-step through the simplified design process I'm about to describe. When you're done you should be able to define your list of art assets for your 3D interiors. You'll also begin creating the objects you need to make an efficient and well-planned game level or walkthrough environment without stress or fear. If done right, this prequalification for art requirements should afford most developers more time at the end of the project to add the finishing polish. This polish is often forgotten or omitted in most final 3D environments because of mismanaged production schedules and loss of time.

As level designers, every level we design and create requires us to determine where we are now, where we are going to end up, and what we will need in the process of getting there. To map this properly requires organization and planning. For S.W.A.T. 3 we managed to achieve a photorealistic look because of much preplanning. We modeled, textured, and designed our lighting schemes using Worldcraft and then loaded information into a proprietary engine. A small team of artists took direction and design recommendations from scaled floor plans which I helped create and then followed certain established modeling and lighting techniques. These dimensional floor plans and architectural drawings were based on the game designer's design document, which sometimes referenced existing real-world buildings and environments.

Whether you are outfitting a room, a laboratory, or the inside of a space station, the process for determining your needs is the same. Preplan the spaces visually in order to understand the architectural features. These features include windows, doors, fireplaces, stairs, columns, an air lock, or a landing platform. These might dictate what kind of furniture will actually fit before you create the pieces. Also keep in mind that furniture is designed to fit people's lifestyles. Each furniture piece is designed to support a particular activity. To visually preplan, I recommend creating a rough layout or floor plan on grid paper and then recording each room using one square for each foot measured. You must establish a working scale for the in-game world and use this when creating and arranging the content for each room. You must also know the measurements of all the

major pieces before creating them and at the same time be ergonomically sensitive, especially in first-person game levels where the characters interact with their environment at close camera view.

The first thing that must be done for each room is to determine the limitations of the space. If there is no physical size limitation, then set one. You can always increase the size later if necessary. Setting self-imposed limitations is still better than not having any at all. Before we can place furniture and other 3D items in a room or level we must understand any technical constraints, including camera limitations, pathfinding capabilities, how many polygons or objects can be displayed in a given area, and the minimum frame rate we need to sustain while traversing that area in the level. Every asset we place in the level directly impacts performance during run time. This kind of information is often difficult to get, because the programmers on the project may not have established these parameters early on in the development cycle, often leading to changes in the level later on. Be persistent in getting the most accurate information as soon as possible. Doing so will prove to be very beneficial in the months that follow.

Next we have to define the purpose of the space and who occupies it. We must then determine the lifestyle and activities of the occupants. Based on these room activities, we can now identify the furniture assets needed to fit the space and support those activities. If they do not already exist, then a list must be made of new art assets that require design and modeling time. We now have the task of placing the chosen items into the room represented on paper. This allows us to continue to flesh out our design without eating up the artist's valuable time creating models that represent pieces to which we haven't fully committed. They should all be at the same scale as our drawing so that if they fit in the plan, they will fit in our room. Finally, we determine all interaction the occupants will have with the space or its furnishings before creating the models.

Now that we know what we need, let's take a look at what we want. A game designer's or a level designer's wants and desires are an expression of who they are and what their product is to become. There is no right or wrong in this; everything desired here is totally legitimate. However, there are situations where what we want, we don't need, and what we need, we don't want. The important thing to determine is what assets are desired and how, yet still solve our in-game problems. Since we've prequalified all the pieces of content in the steps above, the decision-making process now becomes much less painful and costly.

Grow with the Times

It's a tremendous challenge for me to create 3D environments for a game level. To be able to continue to do so, however, experience tells me that I must continue to evolve my process for creating and designing the spaces for such worlds. Ever-improving technology, gameplay demands, player expectations, and shorter development cycles all guarantee that level designers and 3D artists will need to mature their processes for creating attractive virtual destinations to play in. You should need no more motivation than improving yourself as a game artist and what you have to contribute to your team. For me, the alternative is unthinkable. 🎨

Photorealistic Terrain Lighting in Real Time

NATY HOFFMAN | In 1997, after seven years as a microprocessor architect at Intel (where he was the lead architect for the Pentium processor with MMX chip and involved in the MMX, SSE, and SSE 2 instruction set extension projects), Naty took the leap into the game industry and full-time software engineering. He has since been coding up a storm at Westwood Studios, where he has been working on EARTH AND BEYOND as a computer graphics and optimization specialist.

KENNY MITCHELL | Kenny is one of an increasing number of professionals entering game development with a strong academic background. His Ph.D. thesis, "3D Database Environments," introduced the use of real-time 3D graphics on consumer PCs for database visualization. He entered the game industry in 1997 at VIS Interactive plc, where he developed voxel and NURBS real-time rendering technologies. Kenny is director of 3D computer graphics software engineering at Westwood Studios, where his responsibilities include research and development of cutting-edge 3D graphics.

The great outdoors: rolling hills, majestic mountains, sun-drenched plains. An increasing number of games are taking place in outdoor environments, but getting them to look like the view out your window (if you have a house with a nice view) or a scenic postcard (if you don't) is not easy. Outdoor scenes are very complex visually, which makes them hard to render realistically, especially at the high frame rates required for games.

Modern terrain engines (running on powerful graphics cards) are getting pretty good at handling the geometric complexity — all the triangles needed to render those ridges and ravines, erosion lines and canyons — but that just isn't enough. After all, when we look at something in real life, what we see is the light reflected from it. And outdoor scenes have complex lighting, which is a major contributor to the visual intricacy that we find so pleasing (and which makes that house with the nice view so expensive). Getting this right for a single time of day is hard enough, but what if the time of day changes in your game? It would be nice to capture those subtle shifts of light and shadow.

In this article, we present two different methods for achieving these effects in real time. One or the other may be a better match for your game, depending on how you construct the game environments.

Light, Radiance, and Irradiance

Light is electromagnetic energy which radiates through space in all directions — we need a specific physical quantity that describes the intensity of a single ray of light. Luckily, there is such a quantity, called radiance.

To understand how radiance is defined, let us look at a patch of surface (see Figure 1). This patch gives off light (either by glowing or reflection, it doesn't matter) from every point and in all directions, as represented by the red arrows. This light can be measured as a certain amount of energy emitted every second — in other words, as power, which is measured in watts. If we are interested in the light emitted from a specific point on the surface, we can't use power, since the power emitted from a zero-area point is zero. But we can use power per area (measured in watts per square meter), which is represented by the blue arrows. This varies from point to point on the surface. Finally, we are interested in the light emitted in a specific direction from that point. Power per area is useless for

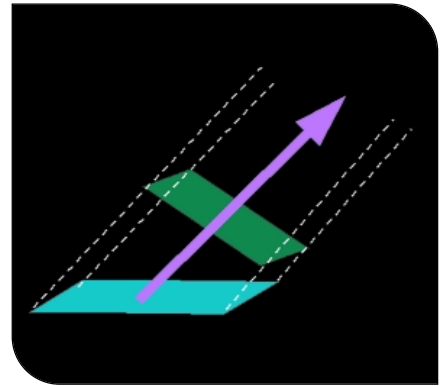
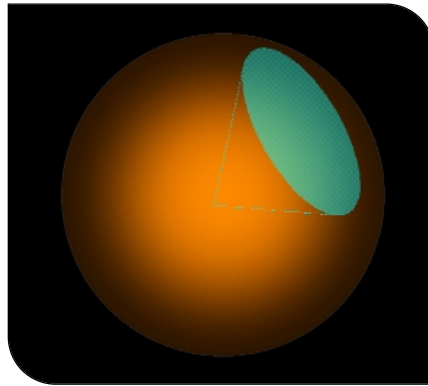
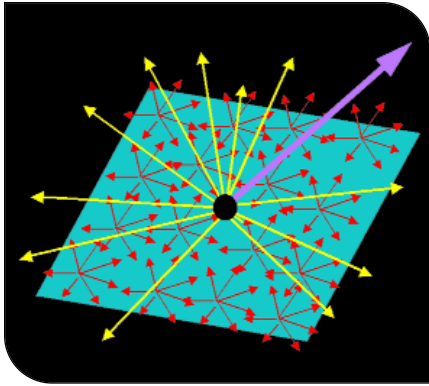


FIGURE 1 (left). Light radiating from a surface. FIGURE 2 (center). A solid angle, measured in steradians. FIGURE 3 (right). Projected area.

this, but we can use power per area per solid angle (solid angles, shown in Figure 2, are the 3D extension of angles and are measured in steradians), which is radiance, as represented by the purple arrow. To be specific, radiance is power per projected area per solid angle. The projected area is the area projected in the direction of the ray (see Figure 3), which is just the area times the dot product between the surface normal and the light ray. It is important to define radiance this way so that the intensity of the light ray is not dependent on the direction of the surface, or even whether there is a surface at all. The units of radiance are watts per meter squared per steradian.

Radiance is a spectral quantity; it can have a different value for each wavelength in the electromagnetic spectrum. For computer graphics we usually just look at three frequencies (red, green, and blue). In real life, radiance has a very large range — the radiance of the sun is more than a million times that of a dark shadow. However, in graphics we are usually limited to a small range, so we will pick a maximum radiance that we can handle, define it as 1.0, and scale everything accordingly. This scale factor is somewhat arbitrary and can vary from scene to scene.

Another important quantity for measuring light is irradiance. The irradiance at a point p , $E(p)$, is simply the total of all the incoming radiance in all directions at p . This total is usually calculated via integration over all incoming directions in the hemisphere around the surface normal (see Figure 4). Irradiance is power per area (not projected area, since it's tied to a surface), and measured in watts per meter squared. Since radiance is defined using projected area, the integration needs to convert from one to the other, which gives us:

$$E(p) = \int_{\vec{v} \in H(p)} L_i(p, \vec{v}) \vec{N}(p) \cdot \vec{v} d\Omega \quad \text{Eq. 1}$$

where $\vec{N}(p)$ is the normal vector at p , $H(p)$ is the hemisphere of outgoing unit vectors \vec{v} centered on $\vec{N}(p)$, $L_i(p, \vec{v})$ is the incident radiance from direction \vec{v} into point p , and $d\Omega$ is the infinitesimal solid angle used in integration.

Why is irradiance important? For purely diffuse, nonglowing surfaces (which we assume our terrains are), the outgoing radiance $L_o(p)$ from the point p is the same in all directions, and is equal to the irradiance at p times the surface color, $C(p)$, divided by π :

$$L_o(p) = \frac{C(p)}{\pi} E(p) \quad \text{Eq. 2}$$

So if we can find the irradiance, we can compute the radiance from it. Color, like radiance and irradiance, is measured separately for red, green, and blue. However, unlike those quantities, its scaling is not arbitrary. A diffuse color of 1.0 means that the surface reflects all the incoming energy that hits it, a diffuse color of 0.5 means that it absorbs half and reflects half, and so on. For nonglowing surfaces, the color can never be greater than 1.0.

Outdoor Lighting

Since the radiance of a surface point depends on its irradiance, let's look at the irradiance of a terrain point p (see Figure 5). We see several factors that contribute to the irradiance. The sun, which covers a small solid angle (it is about 0.5 degrees across)

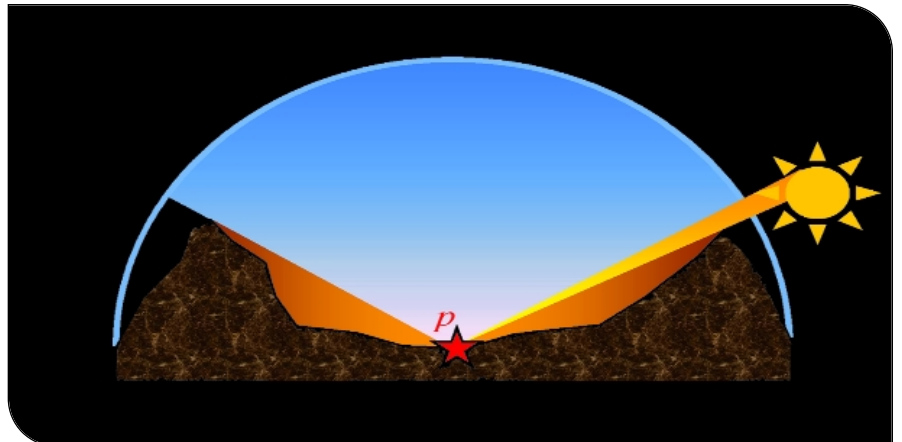
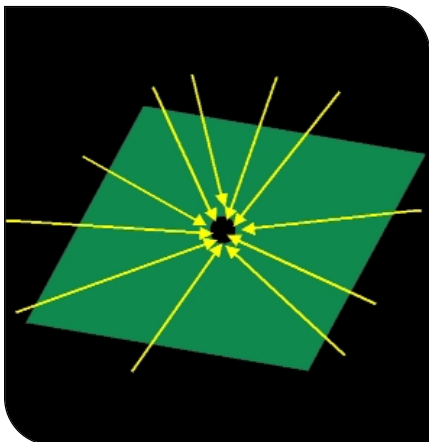


FIGURE 4 (left). Irradiance. FIGURE 5 (right). Outdoor irradiance.

but has a very high radiance, is the most important contributor. In this case, part of the sun's solid angle is hidden by other terrain points. Also important is the sky, which has a lower radiance than the sun but covers a much larger solid angle. This makes it very different from the lights we are used to in real-time graphics and will cause its contribution to look very different from that of a directional light. This contribution is most noticeable in shadows. The remaining incoming directions have interreflections, indirect light reflected from the sun and sky off other terrain points into p . This too is most noticeable in areas of shadow.

A Tale of Two Methods

At Westwood Studios, we ran into the need for realistic outdoor lighting in two different projects: EARTH AND BEYOND and PIRATES OF SKULL COVE. Since the two projects had different needs, we ended up with two very different systems. For both games we wanted high-quality terrain lighting without significantly impacting the run-time performance. However, for EARTH AND BEYOND it was important to have short preprocessing times, a low data footprint, and flexibility in changing the lighting parameters on the fly. For PIRATES OF SKULL COVE the paramount concern was achieving a very high quality overall lighting environment, including the terrain, clouds, and sky. These reasons led us to independently develop an analytical method for EARTH AND BEYOND and a video-based rendering method for PIRATES OF SKULL COVE. We will describe both methods in this article.

Analytical Method

Using this method, we calculate the terrain lighting dynamically as the lighting conditions change with the time of day. This requires us to solve Equations 1 and 2 for every terrain point. We are able to do this in real time by using several simplifying approximations and by doing as much offline precomputation as we can get away with.

We separate the lighting solution into two parts: sunlight and skylight. We calculate each one separately and add the two solutions to get the final lighting result. In our current implementation, we update a lightmap in software as the lighting changes. In the future, we are considering doing the lighting calculation completely in hardware.

Sunlight. Since the sun's solid angle is small, we treat it as a directional light source except for the possibility of partial occlusion (shadowing). Then the sun's direct contribution to the irradiance at each terrain point p , $E_{SunDirect}(p)$, is given by:

$$E_{SunDirect}(p) = O_{Sun}(p) L_{iSun} \vec{N}(p) \cdot \vec{v}_{Sun}$$

where $O_{Sun}(p)$ is the sun's occlusion factor at p ($1 =$ completely visible, $0 =$ completely occluded), and \vec{v}_{Sun} is the outgoing sunlight direction vector.

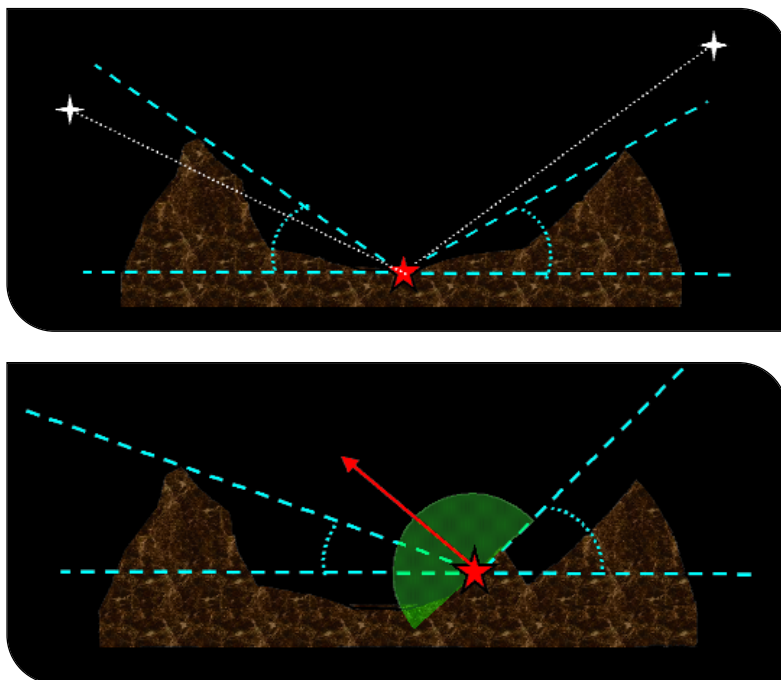


Figure 6 (top). Horizon angles. Figure 7 (bottom). Horizon angle clamping.

We calculate the dot product by using palette normal mapping. In a preprocessing step, we quantize all the terrain normals into a table of 256 normals and store a table index for each terrain point (a process very similar to palettizing a color texture). Then each time we recompute the lighting, we calculate $L_{iSun} \vec{N}(p) \cdot \vec{v}_{Sun}$ for each of the 256 normals. After this step, all that remains for each lightmap texel is to compute $O_{Sun}(p)$ and perform a lookup.

How do we calculate $O_{Sun}(p)$? Again, we use precomputation. We simplify the calculation by assuming that the sun travels in an arc directly over the X-axis. For each texel in the lightmap, we precompute and store horizon angles in the same plane as the sun's arc, thus creating a horizon map (first introduced by Max; see For More Information). The horizon angles (see Figure 6) are stored as 16-bit fixed-point numbers so that $0x0000$ corresponds to 0 degrees and $0xFFFF$ corresponds to 90 degrees. To compute these horizon angles, we scan from each point along the X-axis in both directions, computing elevation angles and remembering the largest ones. We need to scan for a fairly large distance if we want mountains to be able to cast long shadows. This scanning is not expensive in terms of arithmetic calculation, but it is expensive in terms of memory accesses. For this reason it is important to store the height field with the rows along the X-axis so we can scan along rows, as this improves cache behavior.

As the sun's position and color changes, we calculate maximum and minimum angles each frame:

$$\theta_{Min} = \theta_{Center} - \frac{1}{2}\alpha \qquad \theta_{Max} = \theta_{Center} + \frac{1}{2}\alpha$$

where θ_{Center} is the elevation angle of the center of the sun's disk and α is the angular diameter of the sun (about 0.5 degrees, although to achieve pleasing soft-shadow effects, a larger quantity can be used — soft shadow edges also have a useful antialiasing effect on the low-resolution lightmap). $O_{Sun}(p)$ is then calculated by comparing θ_{Min} and θ_{Max} with the appropriate horizon angle ϕ

(measured from the horizontal):

$$\text{if } \begin{cases} \phi \leq \theta_{Min} & O_{Sun}(p) = 1.0 \\ \phi \geq \theta_{Max} & O_{Sun}(p) = 0.0 \\ \theta_{Min} < \phi < \theta_{Max} & O_{Sun}(p) = \frac{\theta_{Max} - \phi}{\alpha} \end{cases}$$

Another way of looking at it is that $O_{Sun}(p)$ is equal to $(\theta_{Max} - \phi) / \alpha$, clamped between 0 and 1. This is simply the fraction of the sun's angle that is unoccluded by other terrain.

There is one important thing to make sure of when using horizon angles. Since irradiance is only measured over the hemisphere around the surface normal, the horizon angles need to be clamped to this hemisphere (see Figure 7). We didn't do this at first, and we had strange visual glitches appear from negative dot products and other peculiarities.

We will ignore the effect of interreflections from sunlight. This is a significant approximation, but it does not overly harm realism, because we will count interreflections from skylight, and the effects of interreflections are most noticeable where the sunlight contribution is small — in shadowed areas. In Figure 8, we can see the result of the sunlight calculation. Note that the shadows are perfectly dark. This is what you would expect to see in an airless environment such as the moon where the sky is pitch black and contributes no illumination even during the daytime. EARTH AND BEYOND (for which we developed this method) actually has such environments, so this is not totally useless, but usually this is not sufficient. Usually, one must also consider and calculate the contribution of skylight.

Skylight. The sky's color varies over its area as well as over time. For simplification, we can divide the sky up into a small number of patches based on elevation angle, on azimuth angle relative to the sun's arc, or both. A small number of patches can capture the sky's illumination nicely — in fact, for our first implementation we have treated the entire sky as one color.

Since the portion of sky "seen" by each terrain point does not change, all we need to do is to precompute the contribution of each sky patch to each terrain point. Namely, what would the illumination of this terrain point be if that patch of sky had a radiance of (1,1,1)? This can be stored in the form of a texture. Then during run time, all that needs to be done is to multiply each sky patch's texture with the current radiance of that patch, add the results together, and we're done.

Precalculating the contribution of direct skylight is fairly simple. Since the sky radiance is constant over each patch, we just solve the integral from Equation 1 to compute the irradiance. If we define the patch simply enough, the integral will have a nice analytical solution that we can calculate directly. But what about the interreflections? We skipped doing them for the sunlight; however, we would really like to have them for the skylight, since this will affect the realism of our shadows. The problem is that such interreflections are a global illumination problem, where the radiance of each point depends on those of many other points, so we need

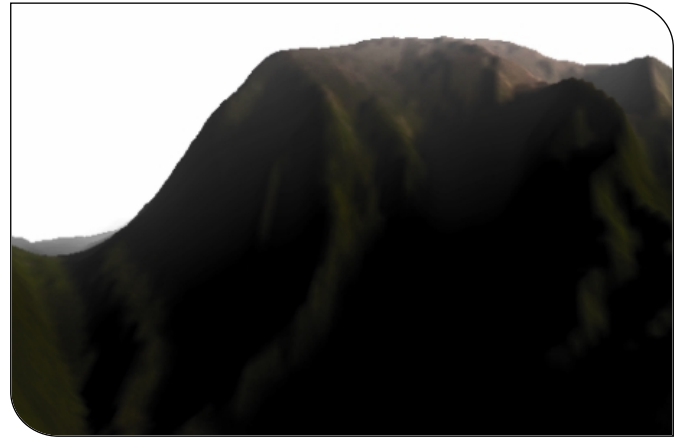


FIGURE 8. Terrain lit by sunlight only.

to perform a very slow iterative process (like a radiosity solve). If the terrains are static and you can afford long preprocessing times in your game, this is a very good solution. Just set up the appropriate patch of sky as an area light source in some tool such as Lightscape, and let it crunch away overnight for a solve. Repeat for each patch and you're done. However, for EARTH AND BEYOND we required much shorter preprocessing times.

Can we solve this directly? If we separate $H(p)$ (the hemisphere of directions around the normal) into $D(p)$ (the subset of directions in which the sky is visible) and $H(p) - D(p)$ (the remaining directions, in which the sky is occluded by other terrain points) and use this separation to expand Equation 1, we get:

$$E_{sky}(p) = \int_{\vec{v} \in D(p)} L_{iSky}(\vec{v}) \vec{N}(p) \cdot \vec{v} d\Omega + \int_{\vec{v} \in H(p) - D(p)} L_{oSky}(x(p, \vec{v})) \vec{N}(p) \cdot \vec{v} d\Omega$$

where $x(p, \vec{v})$ is the terrain point visible from point p in direction \vec{v} .

The $L_{oSky}(x(p, \vec{v}))$ factor is dependent on the lighting solution for other terrain points, which is the problem. Luckily, we ran into a paper by Stewart and Langer (see For More Information) which gives a nice approximation for $L_{oSky}(x(p, \vec{v}))$ and also shows that under diffuse lighting conditions, such as skylight, this approximation introduces very small errors. The approximation is amazingly simple: just assume that $L_{oSky}(x(p, \vec{v})) = L_{oSky}(p, \vec{v})$ for all \vec{v} in $H(p) - D(p)$. This means that the lighting on all terrain points visible from p is the same as the lighting of p itself. Why does this give good results? More details are available in Stewart and Langer's paper, but the basic idea is that for a surface such as a terrain under diffuse lighting, each point tends to "see" points which have lighting similar to itself. The terrain points visible from a point in a dark valley tend also to be in a dark valley, points visible from a bright mountain peak tend also to be bright mountain peaks, and so on.

Applying this approximation results in:

$$E_{sky}(p) = \int_{\vec{v} \in D(p)} L_{iSky}(\vec{v}) \vec{N}(p) \cdot \vec{v} d\Omega + \int_{\vec{v} \in H(p) - D(p)} L_{oSky}(p) \vec{N}(p) \cdot \vec{v} d\Omega$$

Applying Equation 2 gives us:

$$E_{sky}(p) = \int_{\vec{v} \in D(p)} L_{iSky}(\vec{v}) \vec{N}(p) \cdot \vec{v} d\Omega + \int_{\vec{v} \in H(p) - D(p)} \frac{C(p)}{\pi} E_{sky}(p) \vec{N}(p) \cdot \vec{v} d\Omega \quad \text{Eq. 3}$$

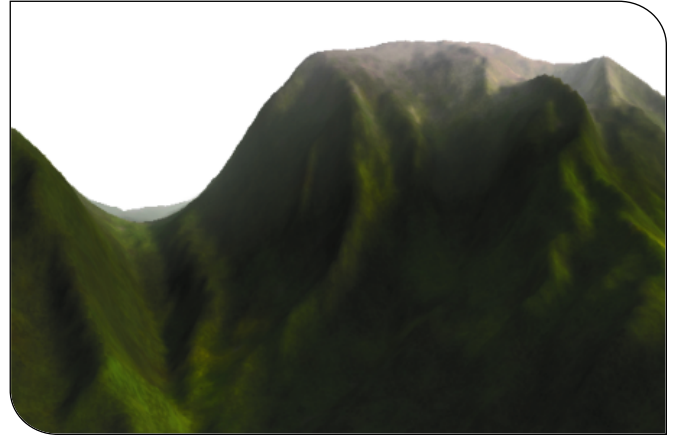
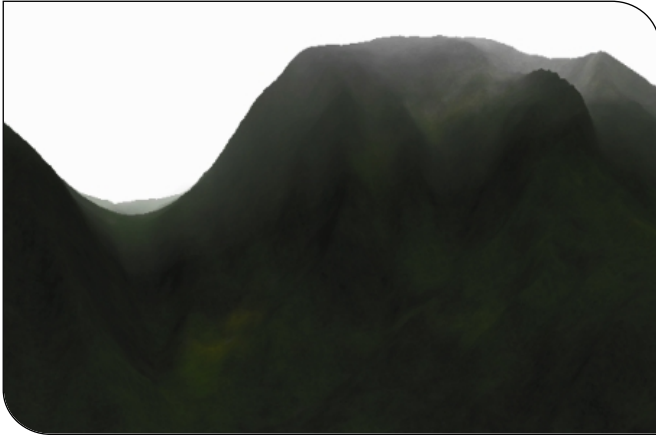


FIGURE 9 (left). Terrain lit by skylight only. FIGURE 10 (right). Terrain lit by both sunlight and skylight.

To simplify the derivation, we will assume for now that the sky radiance is constant over the entire sky. In this case, $L_{i\text{sky}}(\vec{v})$ becomes $L_{i\text{sky}}$ and we can take it out of the integral. It is not difficult to extend the derivation for a sky that is divided into a number of patches, each with its own radiance, if the patches are parameterized carefully.

Note that:

$$\int_{\vec{v} \in D(p)} \vec{N}(p) \cdot \vec{v} d\Omega = \int_{\vec{v} \in H(p)} \vec{N}(p) \cdot \vec{v} d\Omega - \int_{\vec{v} \in D(p)} \vec{N}(p) \cdot \vec{v} d\Omega = \pi - \int_{\vec{v} \in D(p)} \vec{N}(p) \cdot \vec{v} d\Omega$$

Applying this to Equation 3 and using some algebra gives us:

$$E_{\text{sky}}(p) = \frac{L_{i\text{sky}} \int_{\vec{v} \in D(p)} \vec{N}(p) \cdot \vec{v} d\Omega}{1 - C(p) \left(1 - \frac{1}{\pi} \int_{\vec{v} \in D(p)} \vec{N}(p) \cdot \vec{v} d\Omega \right)}$$

This is a nice closed-form expression which takes interreflections as well as direct skylight into account. We can further simplify it by substituting:

$$I(p) = \int_{\vec{v} \in D(p)} \vec{N}(p) \cdot \vec{v} d\Omega$$

which gives us:

$$E_{\text{sky}}(p) = \frac{L_{i\text{sky}} I(p)}{1 - C(p) \left(1 - \frac{1}{\pi} I(p) \right)}$$

Now we need to calculate $I(p)$ — Stewart and Langer also describe how to do this in their paper. This is based on dividing the sky into a number of sectors and using horizon angles (remember those?) to represent $D(p)$. We will use eight horizon angles in the eight compass directions. We already have two horizon angles for the sunlight shadows, so we need to compute just six more in the preprocessing stage. We will not store those extra angles, we will just use them to calculate the skylight texture. We do not need to scan very far for these angles to get good results, which is a good thing because now we need to scan along columns and diagonals of the height field, which is not the best memory access pattern.

Given these eight horizon angles ϕ_i (measured from the vertical this time), the equation for $I(p)$ is:

$$I(p) = \frac{1}{2} \vec{N}(p) \cdot \sum_{i=0}^7 \left(\left(\phi_i - \frac{\sin 2\phi_i}{2} \right) \Delta \sin_i, \left(\phi_i - \frac{\sin 2\phi_i}{2} \right) \Delta \cos_i, \frac{\pi}{4} \sin^2 \phi_i \right) \quad \text{Eq. 4}$$

$$\Delta \sin_i = \sin \left(\frac{\pi}{4} (i+1) \right) - \sin \left(\frac{\pi}{4} i \right)$$

$$\Delta \cos_i = \cos \left(\frac{\pi}{4} i \right) - \cos \left(\frac{\pi}{4} (i+1) \right)$$

Note that the three expressions inside the sum in Equation 4 are the components of a 3-vector. The sum will produce a vector, then the dot product between this vector and $\vec{N}(p)$ is calculated and the result divided by two. Note also that $\Delta \sin_i$ and $\Delta \cos_i$ are constants and can be computed once and reused.

Since each vector being summed depends only on ϕ_i , we can precalculate this vector for each sector and for 256 values of ϕ_i , resulting in a 256×8 table. Note that better accuracy is achieved if we use the tangent of the angle for the table lookup (the tangent of the angle is calculated easily when computing horizon angles).

If we want more than one sky patch, we can assign different sectors to different patches to get two, four, or even eight patches (eight is overkill, though). In this case, a skylight texture needs to be computed and stored for each patch. You don't really need many patches to get good results — currently we are using one, and we intend to try two.

In Figure 9, we can see the result of the skylight calculation. The lighting is very different from the strong directional lighting in Figure 8 and is very good for an overcast day where the sun is completely hidden by clouds and the only light comes from the sky. In this case a gray value for sky radiance would produce good results.

Summary. In the general case, we add the two solutions together, giving the result seen in Figure 10. We get strong sunlight, soft-edged shadows, and subtle variations of light and dark in the shadowed regions resulting from skylight. A time-lapse movie of a day/night cycle using this technique is also available on the *Game Developer* web site at www.gdmag.com.

The equations from the previous sections enable us to calculate the irradiance. To get a lighting (radiance) solution, we need to multiply by the diffuse color and divide by π . One possible way

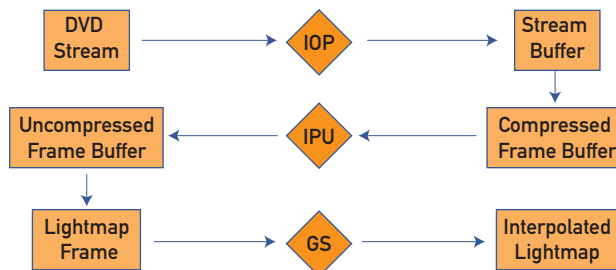
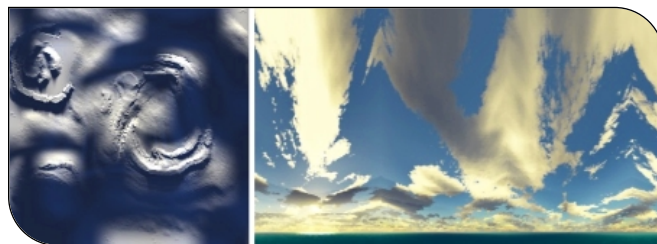


FIGURE 11 (left). Scene from *EARTH AND BEYOND* showing the analytical terrain lighting. FIGURE 12 (top right). Lightmap + skybox — a single frame from a video-based illumination map (VBIM) sequence. FIGURE 13 (bottom right). PS2 video-streaming implementation.

of doing this is to calculate the irradiance divided by π (the division can be put into the various tables so it adds almost no extra cost) and then store the result into a lightmap. This lightmap can be modulated with a color texture by using multi-texture or multi-pass methods. Our terrain engine uses a diffuse color texture that is the same resolution as the lightmap, combined with a repeating detail texture to add noise.

This requires two passes, unless you have a card with three or more texture units. We ended up using a slightly different solution — since we calculate the lightmap in software anyway, we multiply it with the diffuse color texture in software. Then we can draw the terrain with a single pass (light * color map in one texture stage and detail texture in the other).

This makes the lightmap calculation a little more expensive, but we don't care much. The reason is that the lighting changes slowly over time, so we don't need to recalculate the lightmap that often. If we amortize the lightmap calculation over many frames, the performance hit is low. We do this by running the lightmap calculation in a separate thread, but it is possible to do it without multi-threading. We do plan eventually to optimize the lightmap calculations (the inner loop is a very good fit for MMX or the new 128-bit MMX instructions in the Pentium 4), but at the moment this is on the back burner due to the low performance impact.

The precalculation is fast (about six seconds for a 512×512 lightmap on a 450MHz Pentium III) and doesn't require excessive storage (two 16-bit horizon angles and a 24-bit RGB value for each lightmap texel, for a total of 1.75MB for a 512×512 lightmap).

Future directions. Currently, we calculate and upload lightmaps in software. It is tempting to use hardware to generate the lightmaps instead, using texture-blending techniques (such as Direct3D 8 pixel shaders) and multiple passes. The skylight contribution is fairly simple. Each patch's factors can be stored as an RGB texture, and we can just render each texture (modulated with the current patch color) additively to add them up. The sunlight contribution is a little more complicated. Dot-product blending can do the diffuse lighting, and if we drop soft shadows then a simple alpha test can handle the shadowing. If we want soft shadows, we need to perform a subtraction, multiply by a constant, and clamp per pixel —

this should be doable in a pixel shader, and we plan to investigate this possibility.

Local cloud shadows can be simulated by having an additional texture, projected from above, which modulates the sunlight contribution. It would be nice to be able to simulate the sunlight interreflections. Polynomial texture maps (see Malzbender, Gelb, and Wolters under For More Information) appear very promising for achieving this. This addition should increase lighting realism even more.

Video-Based Rendering Method

Think for a minute how long it would take to compute a truly photorealistic terrain rendering, with every detail and nuance represented faithfully. Hours? Days? Accurately simulating photons of light as they interact with particles in the atmosphere without loss of detail could take . . . (insert long duration joke here).

One promising way to approach this level of realism is through the use of image-based rendering techniques (see Debevec in For More Information). In the case of terrain lighting, all the calculations may be either captured from real photographs or computed offline with sophisticated terrain-rendering tools. The results can then be stored as illumination maps and applied as textures in real time with no CPU processing cost.

Figure 12 shows a single frame from a video sequence applied as a single lightmap texture pass to the terrain. In this color image we can see the effects of self-shadowing, cloud shadows, sunlight, skylight, atmospheric blue effects, atmospheric scattering, and haze. All the calculations for these effects are precomputed in the game's asset creation process from raw height-field data using a high-quality terrain-rendering application, Terragen. In addition to streaming lightmaps, a matching sky dome video texture is streamed.

Animating sequences of illumination maps presents us with considerable storage and bandwidth challenges. For example, a typical day/night cycle of 1,000 frames at $512 \times 512 \times 32$ -bit resolution would require 1GB of data. This replaces the task of calculating sophisticated lighting models in real time with the task of performing efficient playback of streaming compressed video



FIGURE 14. Video-based illumination maps in the PS2 game PIRATES OF SKULL COVE.

onto textures in 3D. Let's proceed with how this can be achieved on current hardware.

Video-streaming hardware implementation on Playstation 2. For our upcoming PS2 game, PIRATES OF SKULL COVE, we have the luxury of an image processing unit (IPU), which is a processor dedicated to accelerating the decompression of video data. Importantly, this relieves the CPU entirely of the burden of terrain-lighting calculations, freeing it up to concentrate on game simulation.

Figure 13 depicts the data flow of a single frame of the animation from a compressed video stream stored on DVD to the final rendered lightmap.

Concurrently with the application, compressed frames are streamed into the stream buffer in system memory, using the IO processor (IOP), in much the same way as sound or DVD video is transferred. Once a frame has been fully loaded into the stream buffer, it is copied into the compressed frame buffer, and the next frame begins to load immediately. Compression is necessary to reduce the data rate to within the required limits of DVD media.

The first decompression stage occurs in the IPU, where it processes each frame to produce an uncompressed frame in system memory suitable for upload to video memory. Blending the new frame with the previous one provides the second stage of decompression. This frame interpolation method occurs through the use of the Graphics Synthesizer (GS) and is performed in place by using a frame buffer motion-blur technique to reduce VRAM use (see "Real-Time Full Scene Anti-Aliasing for PCs and Consoles" under For More Information). This second decompression stage is important for a number of reasons:

- It acts as a form of temporal antialiasing between compressed frames, which reduces the number of full frames required for smooth animation.
- The interpolation of frames avoids sudden changes when the looping video jumps to the beginning of the sequence.
- If DVD streaming is held up for any reason, the frame interpolation process will continue unhindered. When the next frame is finally loaded, the same interpolation process will produce smooth "catch up" frames and resume the video sequence as normal.

We can see the results of the PS2 implementation for PIRATES OF

SKULL COVE in Figure 14.

Video-streaming implementations on PC. On the PC, we have an implementation which decompresses the video stream using publicly available software codecs through the DirectShow API. In addition to the performance and quality trade-offs of the various software codecs, the main consideration here is to perform decompression into a system memory buffer and perform double-buffered uploads to video memory to avoid stalls. Two frames from this implementation can be seen in Figure 15, and a time-

lapse video is also available on the *Game Developer* web site at www.gdmag.com.

Hardware-accelerated playback of compressed videos also exists on current PC graphics cards. However, issues with exposure in APIs and hardware conflicts with 3D acceleration are currently blocking an attractive low-bandwidth solution where video data is decompressed directly in video memory. Another possibility is uploading compressed textures. This will reduce bandwidth to the card but not as much as a video stream would.

Illumination map generation vs. real image capture. Ideally, we would capture video-based illumination maps from real video camera footage. One idea for capturing the light field of a real terrain is to place light sensors at regular intervals in a grid at ground level. Another is to extract this information from geostationary satellite image data. Realistically, the logistical problems of setting up these situations make the real image capture method entirely impractical

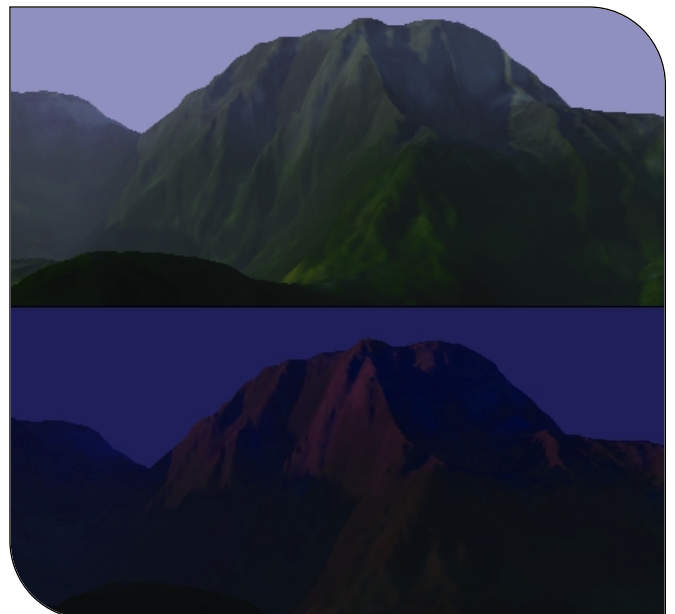


FIGURE 15. Video-based illumination maps on PC.

for game production. Although sky dome/box capture is less problematic, it is just too time consuming to wait for the perfect sunset or the perfect storm. An artist-generated illumination map and sky box could produce the same results in minutes given sufficiently sophisticated rendering tools.

Future directions. With the increased realism of terrain lighting, objects that are placed in this environment must also match the local lighting conditions. Potential solutions range from sampling the terrain illumination map directly beneath the object to streaming irradiance environment maps along with the illumination maps (see Ramamoorthi and Hanrahan under For More Information for more on irradiance environment maps).

Controlled changes in weather conditions could be simulated by branching to alternative video streams. The frame interpolation method would permit smooth transitions between these states.

Using video-based illumination maps in terrain rendering is just one application of video-based rendering in games. With sufficient future hardware support a range of applications will open up for games, such as video-based impostors (see Wilson under For More Information).

In some cases, the sequence of lightmaps as a function of time can be represented by a polynomial texture map (see Malzbender, Gelb, and Wolters under For More Information). This is a promising direction, since the entire sequence can be stored as a small number of coefficients per texel and calculated quickly every frame. This could be useful for other applications of video-based illumination maps as well.

New Directions

Both methods presented here enable outdoor scenes to come closer to the ideal of photorealism by capturing the complexity of the outdoor lighting environment. The analytical method comes close to the quality of global illumination methods while the quality of the video-based rendering method can go as high as you want it to by using whatever offline renderer you like — raytracing, radiosity, photon maps, even real captured movies.

The preprocessing requirements differ between the two methods. The analytical method can perform preprocessing in seconds on a PC and takes up a few megabytes of data. The video-based rendering method can take hours or (in theory) days based on the quality desired, though the use of rendering farms can definitely cut down on the time needed. With the video-based method, the preprocessed data is a video stream, the size of which will vary based on the resolution, frame rate, and codec used.

Since both methods rely heavily on preprocessing, they work best with static scenes. The analytical method can support local changes in geometry by redoing the preprocessing for the terrain region affected by the change. For this to be practical, the affected region should be a very small part of the total scene.

Both methods are examples of new directions in real-time rendering. Increased programmability in hardware will enable us to perform sophisticated global illumination calculations analytically. Video-based rendering and lighting can be used to produce movie-quality effects in real time, and we will be able to use the results of movie production methods in our games. 🎮

ACKNOWLEDGEMENTS

We would like to thank ATI and Nvidia for supplying us with hardware for experimentation and demos, and Hector Yee for his help on the analytical method implementation.

FOR MORE INFORMATION

SOFTWARE

Terragen

www.planetside.co.uk/terrigen

Free to download for noncommercial use. Commercial use is permitted for registered users.

REFERENCES

- Debevec, P. "Pursuing Reality with Image-Based Modeling, Rendering, and Lighting." Keynote presentation at the Second Workshop on 3D Structure from Multiple Images of Large-Scale Environments and Applications to Virtual and Augmented Reality (SMILE2), Dublin, Ireland, June 2000.
www.debevec.org/Publications
- Heidrich, W., K. Daubert, J. Kautz, and H.-P. Seidel. "Illuminating Micro Geometry Based on Precomputed Visibility." *Computer Graphics (Proceedings of SIGGRAPH 2000)*, July 2000: 455–464.
www.cs.ubc.ca/~heidrich/Papers/index.html
- Hoffman, N., and K. Mitchell. "Real-Time Photorealistic Terrain Lighting." *2001 Game Developers Conference Proceedings*, March 2001: 357–367.
www.gdconf.com/archives/proceedings/2001/prog_papers.html
- Max, N. L. "Horizon Mapping: Shadows for Bump-Mapped Surfaces." *The Visual Computer* Vol. 4, No. 2 (July 1988): 109–177.
- Malzbender, T., D. Gelb, and H. Wolters. "Polynomial Texture Maps." To appear in *Computer Graphics (Proceedings of SIGGRAPH 2001)*, August 2001.
www.hpl.hp.com/ptm
- Mitchell, K. "Real-Time Full Scene Anti-Aliasing for PCs and Consoles." *2001 Game Developers Conference Proceedings*, March 2001: 537–543.
www.gdconf.com/archives/proceedings/2001/prog_papers.html
- Ramamoorthi, R., and P. Hanrahan. "An Efficient Representation for Irradiance Environment Maps." To appear in *Computer Graphics (Proceedings of SIGGRAPH 2001)*, August 2001.
<http://graphics.stanford.edu/papers/envmap>
- Schödl, A., R. Szeliski, D. Salesin, and I. Essa. "Video Textures." *Computer Graphics (Proceedings of SIGGRAPH 2000)*, July 2000: 489–498.
www.gvu.gatech.edu/perception/projects/videotexture
- Sloan, P.-P., and M. F. Cohen. "Interactive Horizon Mapping." *Rendering Techniques 2000 (Proceedings of the Eleventh Eurographics Workshop on Rendering Workshop)*, June 2000: 281–298.
www.research.microsoft.com/~cohen
- Stewart, A. J. "Fast Horizon Computation at All Points of a Terrain with Visibility and Shading Applications." *IEEE Transactions on Visualization and Computer Graphics* Vol 4, No. 1 (March 1998): 82–93.
www.dgp.toronto.edu/people/JamesStewart/papers/tvcg97.html
- Stewart, A. J., and M. S. Langer. "Towards Accurate Recovery of Shape from Shading under Diffuse Lighting." *IEEE Transactions on Pattern Analysis and Machine Intelligence* Vol. 19, No. 9 (Sept. 1997): 1020–1025.
www.dgp.toronto.edu/people/JamesStewart/papers/pami97.html
- Wilson, A., M. C. Lin, D. Manocha, B.-L. Yeo, and M. Young. "A Video-Based Rendering Acceleration Algorithm for Interactive Walkthroughs." *Proceedings of ACM Multimedia*, October 2000.
<http://woodworm.cs.uml.edu/~rprice/ep/wilson>

Game Developer's

Salary Survey

How much money should you expect to make as a game developer? How does \$61,403 sound? That was the average of all 1,801 people who responded to salary information in a survey conducted of *Game Developer* magazine subscribers, Gamasutra.com members, and Game Developers Conference attendees.

With the help of research firm Market Perspectives, we sent e-mail to *Game Developer* magazine subscribers in July 2000 inviting them to participate in our survey and received 919 responses. Last November, we e-mailed invitations to all Gamasutra.com members to take the survey online and received 1,953 responses. Then, in March 2001, 1,797 GDC attendees took the survey on-site at computer terminals. Not all developers who participated in the survey answered the salary-related questions, which is why the total sample reflected in the data presented in the following pages, 1,801, is smaller than the total number of respondents. Besides cases where salary data was omitted from surveys, we also excluded cases where the salary was given at less than \$10,000 or greater than \$300,000 or where there was text entered in the salary box that did not represent a salary figure.

The sample represented in our salary survey can be projected to the game developer community with a margin of error of plus or minus 2.29 percent at the 95 percent confidence interval. That means that with the number of respondents in our sample, we can say with 95 percent certainty that the statistics would stay consistent across the entire population.

Another thing we can't measure with these numbers is developers' job satisfaction. If you make \$60,000 per year and work 40-hour weeks, your average hourly rate for the year is \$28.85. However, if you work 40-hour weeks for eight months of the year and 80-hour weeks for four months of the year, your average hourly wage for the course of the year ends up being \$21.63, which is equivalent to \$45,000 per annum for someone who works straight 40-hour weeks at that rate for the whole year. If there's anything driving game developers to endure yet another crunch mode and bear the burden of time spent away from home and loved ones, it's the satisfaction they get from contributing technical sparkle, artistic flourish, or innovative gameplay while bringing a unique form of entertainment to a wider audience. To say nothing of the sheer joy many developers take in actually getting paid to do something they'd gladly stay up all night in their spare room doing on their own time.



How does your salary stack up?

JENNIFER OLSEN | Jennifer is the senior editor of Game Developer and Gamasutra.com.
JILL ZINNER | Jill has been a recruiter for 25 years and has been specializing in the computer games industry for almost 10 years. She is the president of Premier Search Inc., based in Nevada. Her e-mail address is jillz@ix.netcom.com.

PROGRAMMING

Who is a programmer? Our survey considered a “programmer” to be a person who described themselves as an engine programmer, AI programmer, tools programmer, hardware engineer, network programmer, or simply a programmer. It also includes people who have been around long enough to have the title of senior programmer as applicable to any of these job titles. A lead programmer is understood to be someone who is responsible for managing a team of other programmers and scheduling. A technical director or director of development is someone responsible for the overall management of a company’s technology decisions and might manage a single team of programmers at a small company, or a group of leads on various projects at a larger company.

“Programmers have it the best, salarywise. Good games programmers are extremely rare, and even mediocre ones are pretty rare. But a really good programmer, with a history in the industry, can command a huge salary.”

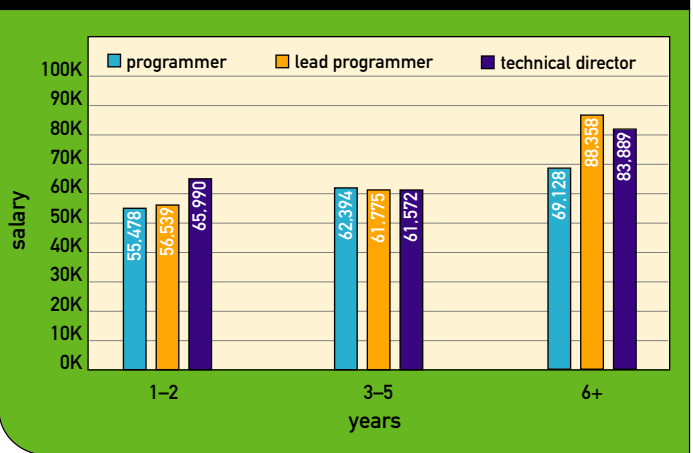
— programmer, California

Clearly, experience pays. It’s also much harder to hire for. If you’re looking for a programmer with at least three years’ game programming experience, you’ve already eliminated more than half of the game programmers out there, 54.3 percent, who have only one to two years’ experience in the industry. You can also expect to pay dearly for a seasoned lead programmer or technical director with six or more years’ experience.

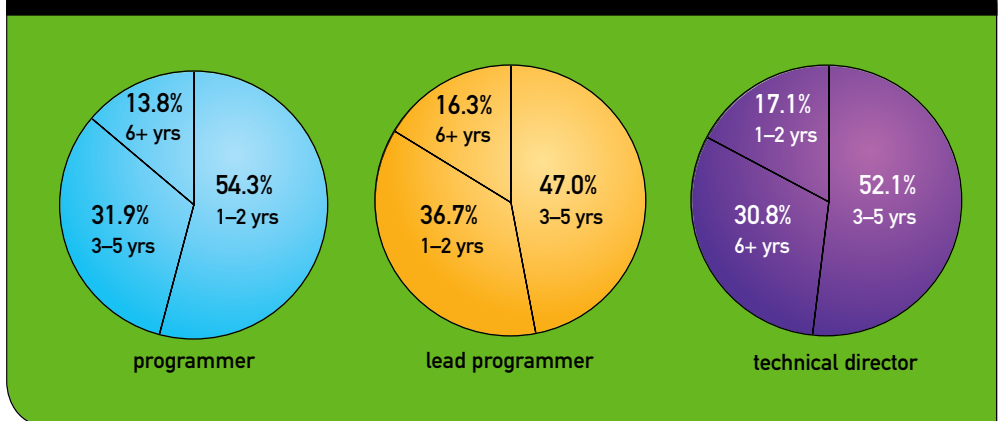
What Employers Want

Programmers are gold. If you’re a programmer who has published some titles, or can show that you have made and completed a game, it proves that you can finish what you start. A lot of developers have problems putting the finishing touches on things. Proving that you can finish what you start is very important to a potential employer. Many people can’t get a job because they have not completed a game, leading to the common catch-22 of first-time job-seekers. Lacking a published title, you should at least show a prospective employer that you can work to create something others have fun playing. Many companies ask developers for code samples. Your best bet is to have your résumé and a disk with code samples available, preferably code samples from a working game. — Jill Zinner

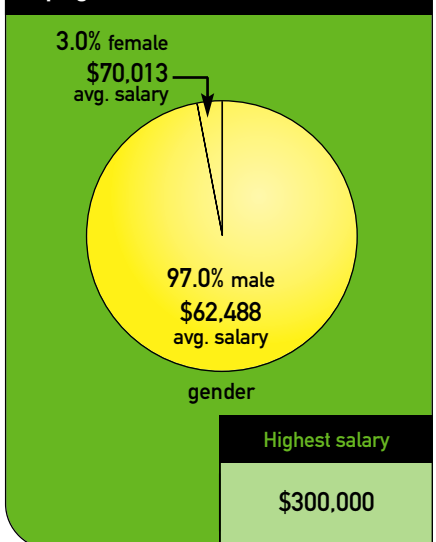
Programming salaries per years of experience and position



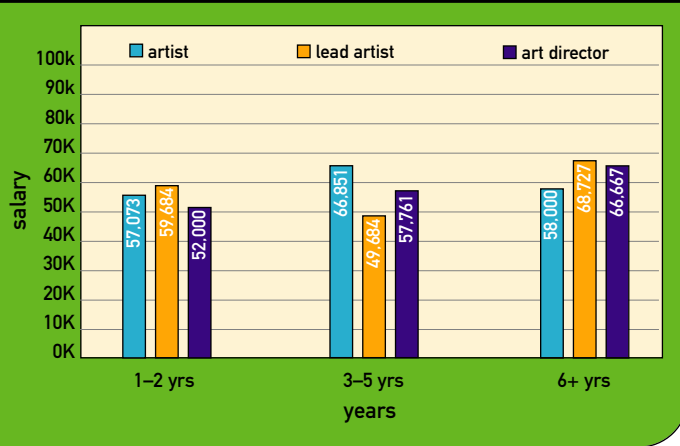
Years experience in the industry



All programmers



Art salaries per years of experience and position



ART

Who is an artist? We received salary information from artists who defined themselves as animators, 3D artists/modelers, and 2D artists/texturers. We grouped lead artists and lead animators under the heading of “lead artists,” people who manage a team of artists and who construct schedules and help establish the artistic direction and feel of a game. Art directors might fill this same function at a smaller company, while at a larger company art directors might oversee a range of different products or manage the aesthetic of a product line with other leads under them.

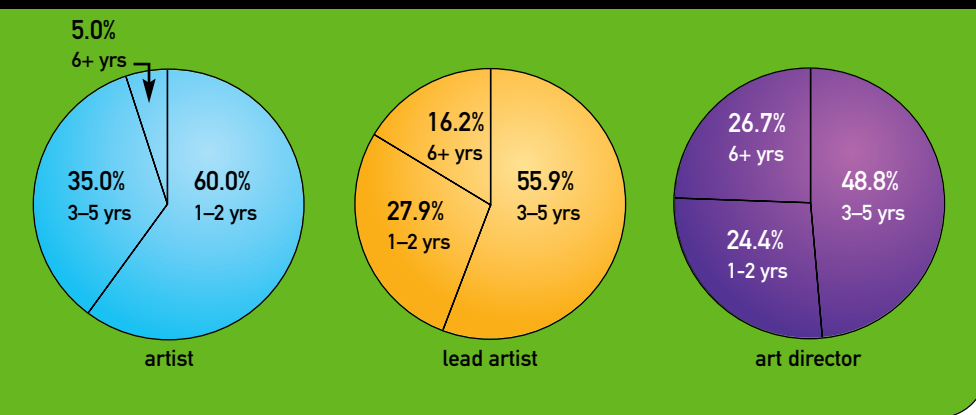
Unlike in other game development disciplines we looked at in this survey, artists’ salaries seemed relatively scattered across years of experience and level of responsibility. This may suggest that salaries offered to artists are more subjective than salaries offered to technical people,

whose skills are more quantifiable in conventional terms. Another surprise is that while artists are widely assumed to earn less overall than their counterparts on the programming side of the fence, artists in some categories are actually commanding higher salaries, most notably at the entry level.

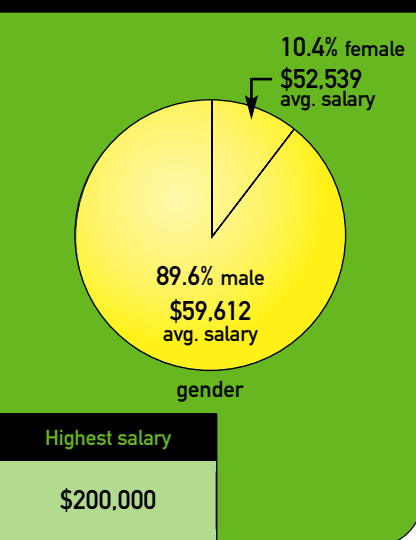
“Salaries are subject to the laws of supply and demand, and most people seem to be mature enough to understand this and don’t let it become an interpersonal issue.”

— animator, Washington

Years experience in the industry



All artists



Staying competitive. Just as programmers must work to remain on the competitive edge of technology, so must artists continue to adapt and evolve with changing technology in game development. For the same reason that programmers stopped doing art when we exited the 8-bit era, the creative demands on professional artists will continue to mount as polygon counts, fill rates, and available texture passes increase steadily with every generation of hardware that hits the street. Demand will no doubt accelerate for artists who keep up with the latest software and technologies. For an art director awash in demo reels, artists who can demonstrably manipulate subdivision surface patches, massage intricate facial-cap data, write time- and labor-saving scripts for a 3D art package, help construct an effective art path, and communicate productively with their programming, production, and design teams will no doubt be rewarded for their expertise.

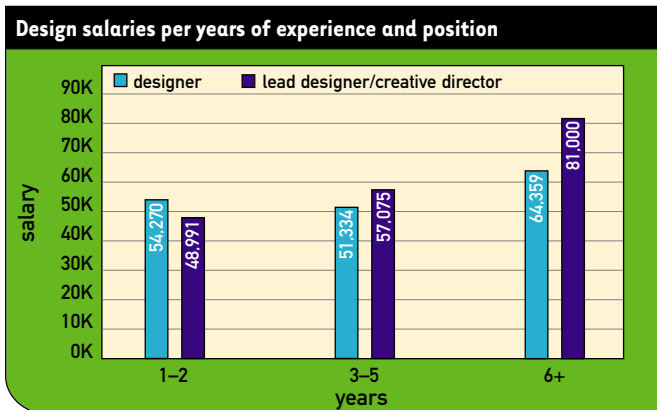
There is also growing demand for art techs. Currently, this position often falls to whichever programmer on a team has the strongest grasp of art software, or whichever artist has an unusual proclivity for understanding and applying technology. It is a unique and increasingly critical combination of skills, one for which experienced art techs can expect to be compensated well in the years to come, whether they come from the programming or art side.

DESIGN

For the purposes of this survey, we considered a “designer” to be a game designer, a level designer, or a writer. In smaller companies, one person might fulfill such a role, whereas larger projects or companies might have different people assigned to these specialized tasks. A lead designer or creative director is someone in charge of coming up with overall gameplay concepts and overseeing the design process, writing and maintaining design documents, and managing a design team to implement their creative vision. For designers, experience is an important factor in commanding higher salaries, especially for designers with six or more years’ experience.

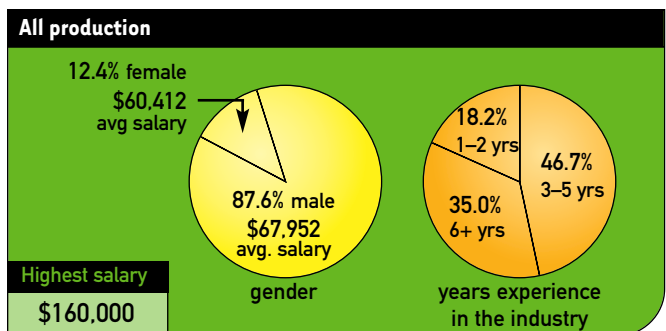
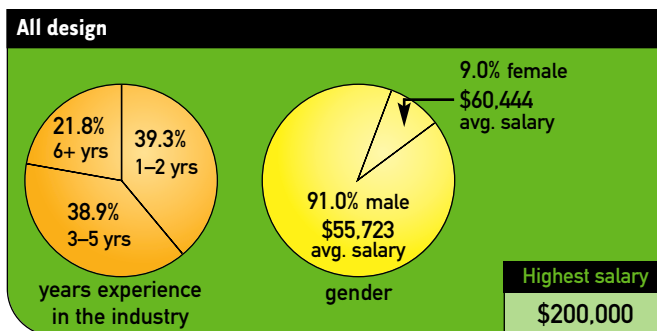
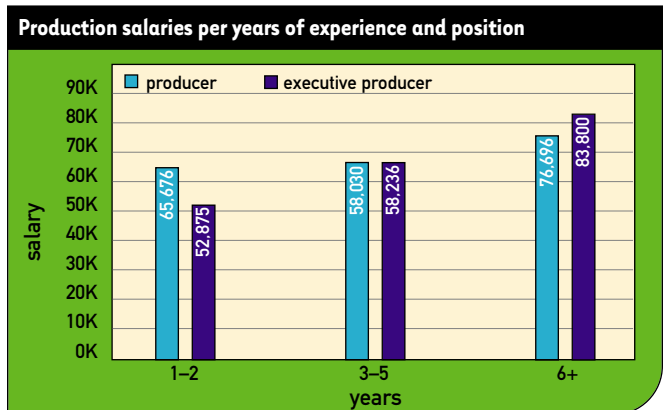
“I was definitely surprised at how little money I was offered as a starting salary. Luckily I stuck it out, and my salary grew at a substantial rate.”

—lead designer, Wisconsin



PRODUCTION

Forever fighting off the image of the coffee-cup-toting clipboard-wielder who leaves work right at the stroke of five, producers have some of the most eclectic job responsibilities in game development. For the purposes of our survey, we considered a “producer” to be anyone who described themselves as a producer, associate producer, or project lead/manager. These people have a range of functions: planning and managing the QA process, setting up motion capture shoots, communicating with the publisher, managing the overall flow of game assets, planning localization, managing the overall project schedule, and essentially doing anything else that will help ensure the game is completed on time. People who describe their jobs as executive producers typically have more production experience, or might oversee more than one product or producer at a time. Often they have come up through the ranks with steadily increasing responsibility.



The QA Breeding Ground

For a huge percentage of the game industry, the quality assurance department is the training ground. Engineers who are self-taught often come through QA; designers and producers almost always come from that environment. They start in customer service and work into QA, and then have a choice of going into development or marketing. If they choose the development path, they usually choose either design or production.

In the old days, game design almost always came from the programmers, who taught themselves to program by trial-and-error while pursuing their idea. These

days, though, many designers come from QA or customer service, where they have to find bugs and work with the developers to fix them. This process, and not recreational programming, brings them into the process of design, and development in general. Producers typically grow the same way. The QA or customer service person has to work with the producer who is a liaison to the development team. Pretty soon this person is assisting the producer and gradually evolves into a full producer after a few promotions. It’s hard to leave a company while still in QA and find a job as a producer, designer, or programmer elsewhere. The first promotion almost invariably must come from within the company. — Jill Zinner

AUDIO

The audio function in game development is so varied and so arcane to many other developers, is it any wonder so many professionals voluntarily assume the simple moniker of “audio guy”? Audio professionals might be responsible for audio engineering, sound effects design, musical composition, and working with the producer recording and editing voice-overs. It has long been customary for game developers to turn to outside contractors for their game audio needs, but as more and more companies are taking on multiple projects, more are finding the benefit of having at least one full-time audio professional on staff.

Audio is another discipline in which experience clearly pays for our survey respondents. With experience, you can show not just your creative talent in a shipping product’s audio, but also demonstrate through references on past projects that you know what it takes to get a complicated and very critical job done (often with varying degrees of direction from producers and designers) in the invariably tight timeframe required by the project.

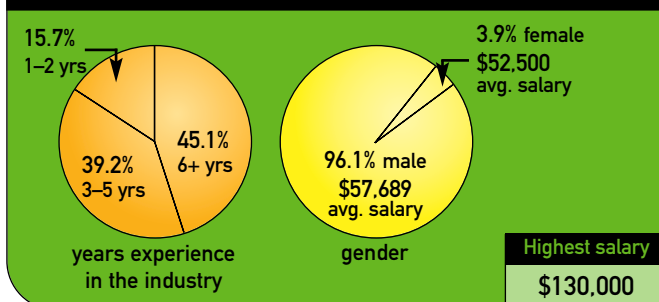
“I started working in the game industry 14 years ago. There was no money back then, we were just kids who were doing it for fun. So I guess having any salary is an improvement over those days.”

— sound director, California

Audio salaries per years of experience



All audio



Call for Writers

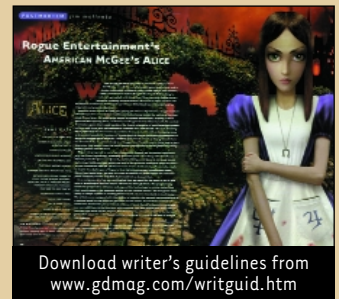
Dear Professional Game Developers,



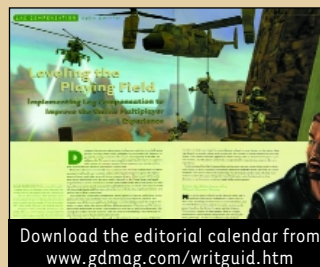
Write Soapboxes, Postmortems, or feature articles

If you’re technically astute and have a way with words, *Game Developer* magazine needs you. We’re looking for feature articles on programming (graphics, AI, networking, and so on), art and animation techniques, game/level design, audio composition and technology, testing and QA, business/legal issues, and other relevant subjects. We’re interested in development on a wide variety of platforms, including the PC, Macintosh, game consoles, arcade machines, the web, and handhelds. We also publish Postmortems on completed games, and Soapboxes on various issues pertinent to game developers.

Game Developer is the game development industry’s community forum. We reach over 90,000 readers each month (directly or by being passed along). Take this opportunity to share your hard-earned knowledge with the entire game development community! Please send your article abstract, outline, or completely wacky idea to: mdeloura@cmp.com.



Download writer’s guidelines from www.gdmag.com/writguid.htm



Download the editorial calendar from www.gdmag.com/writguid.htm

Thanks,

Mark DeLoura
Editor-in-Chief
Game Developer

GameDeveloper

OTHER TRENDS

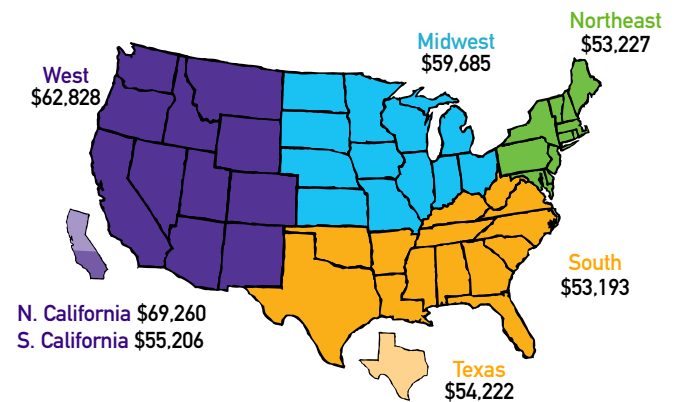
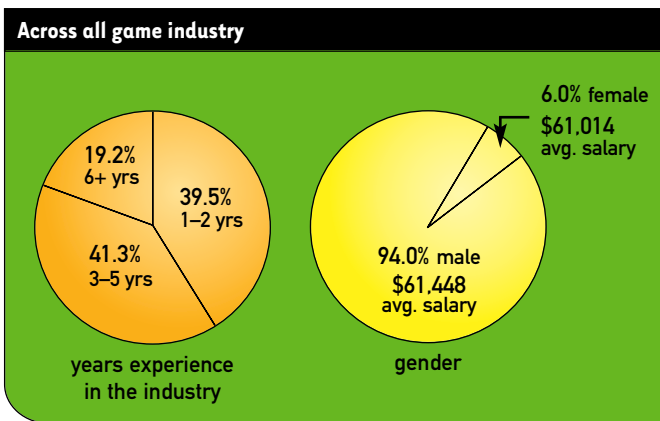
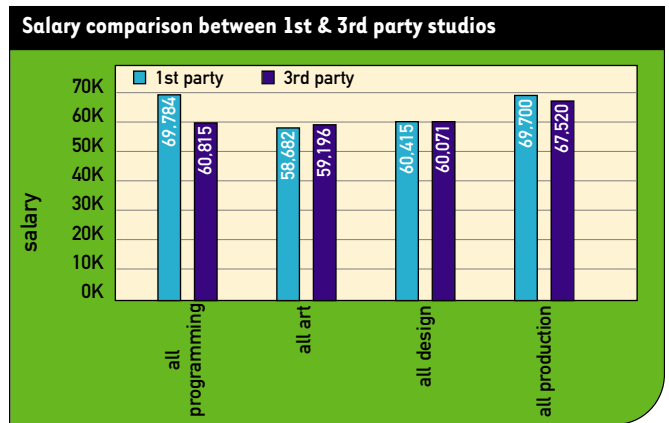
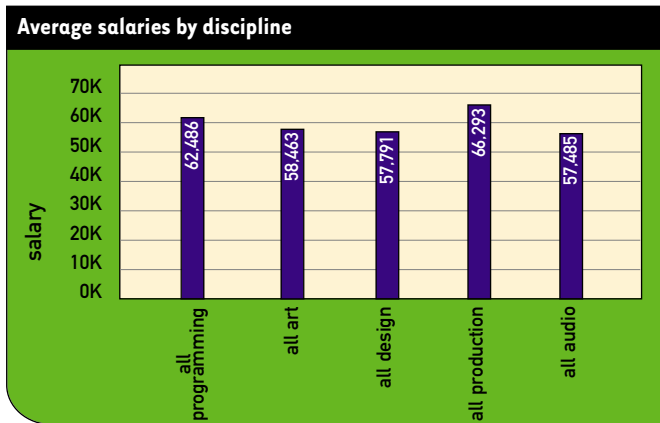
The laws of supply and demand prevail in game development salaries. Higher salaries generally go to those who require more specialized skills and hence are harder to hire for, such as programmers, than in areas where supply exceeds demand, such as in art and design positions. However, the disparity in pay is not as gaping as those looking from one side of the fence to the other might have suspected — only 6.9 percent difference overall between programmers and artists of all levels of responsibility and years of experience, and 8.1 percent between programmers and designers.

Realities of supply and demand also help fuel differences in regional game development salary averages. For example, Northern California, which hosts a booming high-tech industry with a chronic shortage of skilled technical workers, offers higher salaries than

regions where the competition for available qualified talent is not as stiff.

Only 6.0 percent of our survey's respondents were women, and their salaries were 0.7 percent lower overall than their male counterparts (or 99.3 cents on the dollar). This disparity is far better than women fare in the national average of just 76.5 cents on the dollar compared to men in 1999, as reported by the Census Bureau and the Bureau of Labor Statistics, an arm of the U.S. Department of Labor.

Who makes what and why is just as controversial in game development as it is in other industries. Indifferent economic principles are at work alongside human desires for equity and fair recognition for one's contributions. For many game developers who couldn't imagine doing anything else for a living, however, compensation is just icing on the cake. 🍰

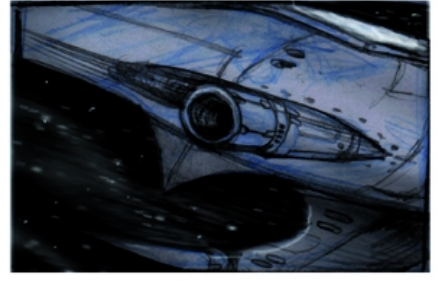
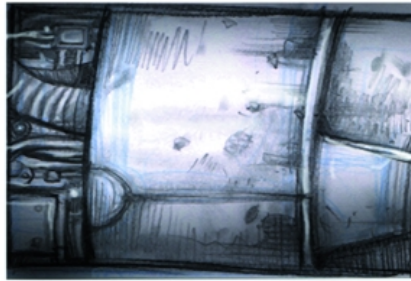
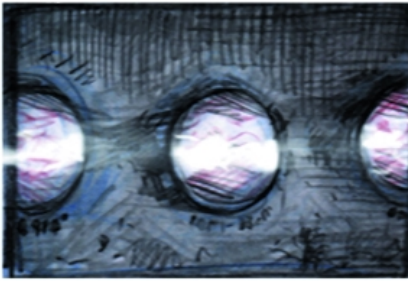


Should I Stay or Should I Go?

Development teams for market-relevant games can require anywhere from six to 35 people. Many senior people have reached the ceiling salary for their position. New technology is eliminating some positions, changing others, and creating new needs. Experienced developers are now finding that they have to make some serious decisions about their career. They might consider taking less money or relocating to a different part of the country where the cost of living is more reasonable.

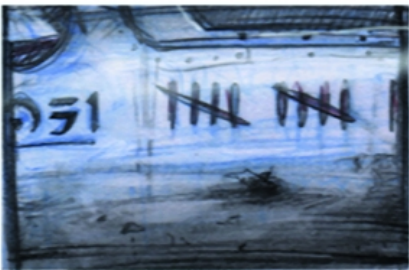
New studios are starting up everywhere, and so jobs are cropping up all over the country. These small studios work for the big publishers. The publishers are trying to cut the cost of making their products, so they look to outside developers to make the cost

of making a game more reasonable. The general trend of our industry today is the migration of all the great talent out into these new studios. Generally, game developers change jobs about every two years or at the end of a game cycle. Terrific programmers, sometimes whole teams, get disillusioned with the companies where they work and strike out to do it on their own. What entices people to make such a career shift? One factor is that the cost of living is so different all around the country. People who want to buy a house or raise a family are looking for jobs at game companies where the cost of living is lower and the pace is slower. Also, many of these outlying studios are trying to get back to the basics of making games, fostering a culture which seems attractive to many developers coming from large, corporate environments. — Jill Zinner



PROJECT EUROPA

LucasArts' STAR WARS STARFIGHTER



W

ork on Project Europa — the internal codename for the development effort that would eventually metamorphose into STAR WARS STARFIGHTER — began in earnest in April 1998. A small crew of programmers, headed up by director Daron Stinnet, began preproduction work on a *Star Wars: Episode I* PC title that had grand ambitions. As one of LucasArts' great unsung talents, Daron had previously led the DARK FORCES and OUTLAWS teams to much critical and commercial success. Now, following in the footsteps of Larry Holland's X-WING games, Europa was to bolster LucasArts' presence in the space-combat genre and support the new film franchise. While embracing much of the X-WING series's simulation-oriented aesthetic, the team also wanted to deliver the visceral, sweaty-fingered arcade experience that we were starting to see in early builds of ROGUE SQUADRON.

During the early months of 1999, a well-known designer who was in the market for a new lead programmer and lead level designer for his company's overdue project secretly approached two members of our team about the possibility of jumping ship. Although obviously conflicted, the allure of working with a famous industry heavyweight proved too tempting, and within a few short weeks we had lost our main graphics programmer and level designer. Shaken but undeterred, we were determined to make the best of a bad situation, but three months later the project suffered another blow when we lost our second graphics programmer.

This was Europa's darkest hour. The technology development was progressing slowly, and our inexperienced programming staff was still climbing the C++ learning curve. As lead programmer, this predicament was largely of my own making. I had joined LucasArts from outside the game industry, where I was accustomed to a corporate R&D environment that valued solid engineering and extensible software architecture over quick solutions that were perhaps less elegant or flexible. Now, with little to show but a creaky Glide-based graphics engine and no graphics programmer, we were at a loss as to what to do next.

As if things weren't bad enough, we were also floundering on the game design side of the fence. Although we had a lot of excellent concept art, few of us had a clear idea about exactly what type of game we were making. We were painfully starting to discover that while it is easy to characterize a title as being a cross between ROGUE SQUADRON and X-WING, it's another thing completely to describe what that actually means.

At this point two events occurred that I'm convinced saved the project. Our multiplayer programmer, Andrew Kirmse, who had already proven himself as a remarkably capable technologist, teamed up with two of our other programmers to create a graphics-engine "tiger team," a small subteam dedicated to attacking a single task with unwavering focus. In just a few months the three of them delivered a brand-new OpenGL-based engine that was far better than anything we had built previously.

Shortly after the new graphics engine came online we also found the solution to our game design woes. Tim Longo, who had recently helped complete INDIANA JONES AND THE INFERNAL MACHINE, joined the team as our lead level designer. The change was immediate and profound; five other level designers joined the project at about the same time, and now we had the foundation for a thriving, collaborative design process. Daron worked with Tim and the other level designers on an almost daily basis, systematically identifying areas of the game design that were incomplete and working together to come up with concrete solutions.

By the end of 1999 the project had performed a 180-degree turnaround, but there was one more significant twist in the road awaiting us. Sony had turned the game industry on its ear with the formal announcement of the Playstation 2 that year, and every major development house was furiously rewriting business plans to accommodate support for the new platform; LucasArts was no exception. The biggest problem for the company was that we wanted to have a title close to the system's launch, and



GAME DATA

PUBLISHER: LucasArts

FULL-TIME DEVELOPERS: Approximately 40 at the height of production

LENGTH OF DEVELOPMENT: 30 months

RELEASE DATE: February 2001

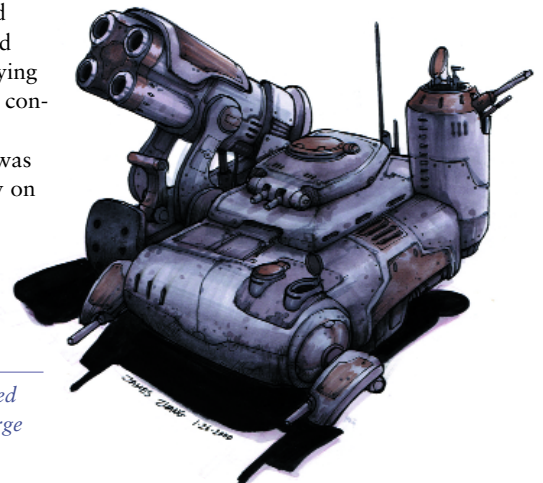
PLATFORM: Playstation 2

HARDWARE USED: 700MHz Pentium III's with 256MB RAM, GeForce 256, PS2 tools

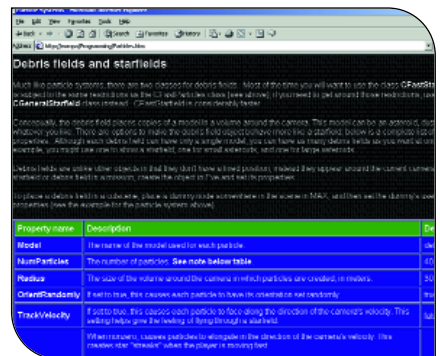
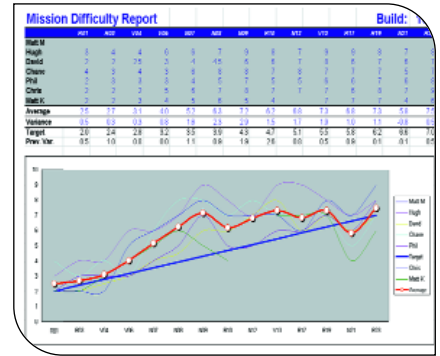
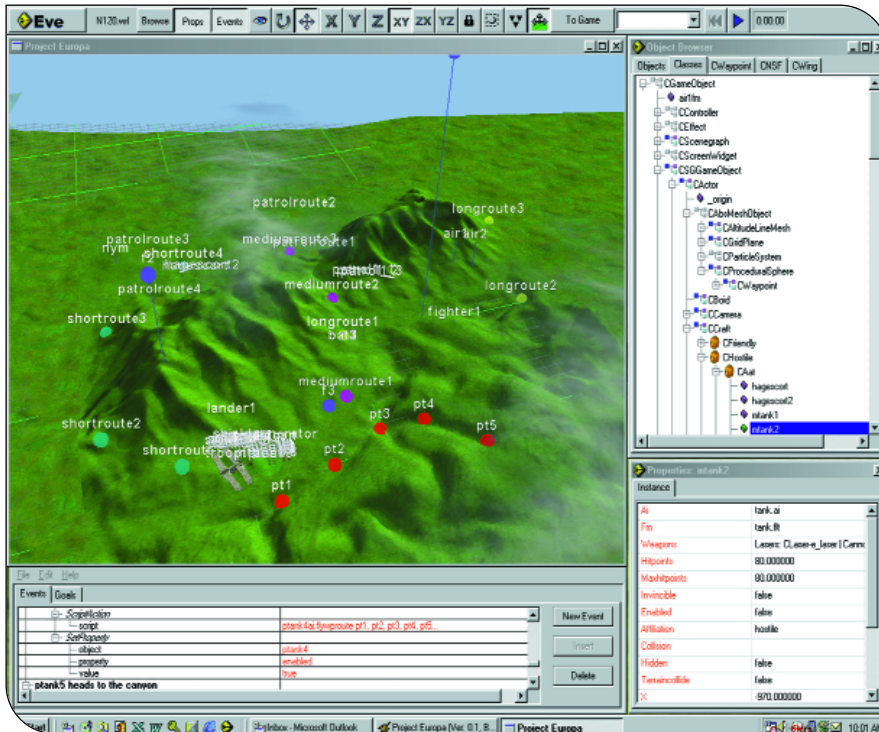
SOFTWARE USED: Windows 2000, Microsoft Visual C++, Metrowerks for PS2, 3D Studio Max, Softimage, Photoshop, Bryce, Visual SourceSafe, Perl, AfterEffects, Premiere

TECHNOLOGIES: Eve level design tool, Miles Sound System, ObjectSpace STL, Macromedia/Secret Level Flash, Planet Blue's Tulip for prerendered cutscene lip-synching

LINES OF CODE: 301,000 including tools



CHRIS CORRY | *Chris was the lead programmer on STAR WARS STARFIGHTER. He joined LucasArts three years ago after a seven-year stint in the "real world" helping to manage large R&D teams building distributed object-oriented architectures.*



LEFT: The Eve level design tool was a critical part of STAR WARS STARFIGHTER's success. TOP RIGHT: An example of the statistics that Daron tracked during the game's development. BOTTOM RIGHT: A page from the programming section of the STAR WARS STARFIGHTER internal web site.

Europa was the only project far enough along to be a serious candidate. The thought of throwing the PS2 into the mix made many people very uncomfortable, but when we were able to port all of our nongraphics code in a single 48-hour period, senior management became convinced. The rest of the project was an exciting and manic blur of activity. Early in 2000 we hit the "snowball point," that period when all of a sudden the tech falls into place, the art production paths are running on all cylinders, and the team is seeing exciting new gameplay on an almost daily basis. From then on, STAR WARS STARFIGHTER was indeed like a runaway snowball, picking up momentum and new features almost as fast as we could think of them.

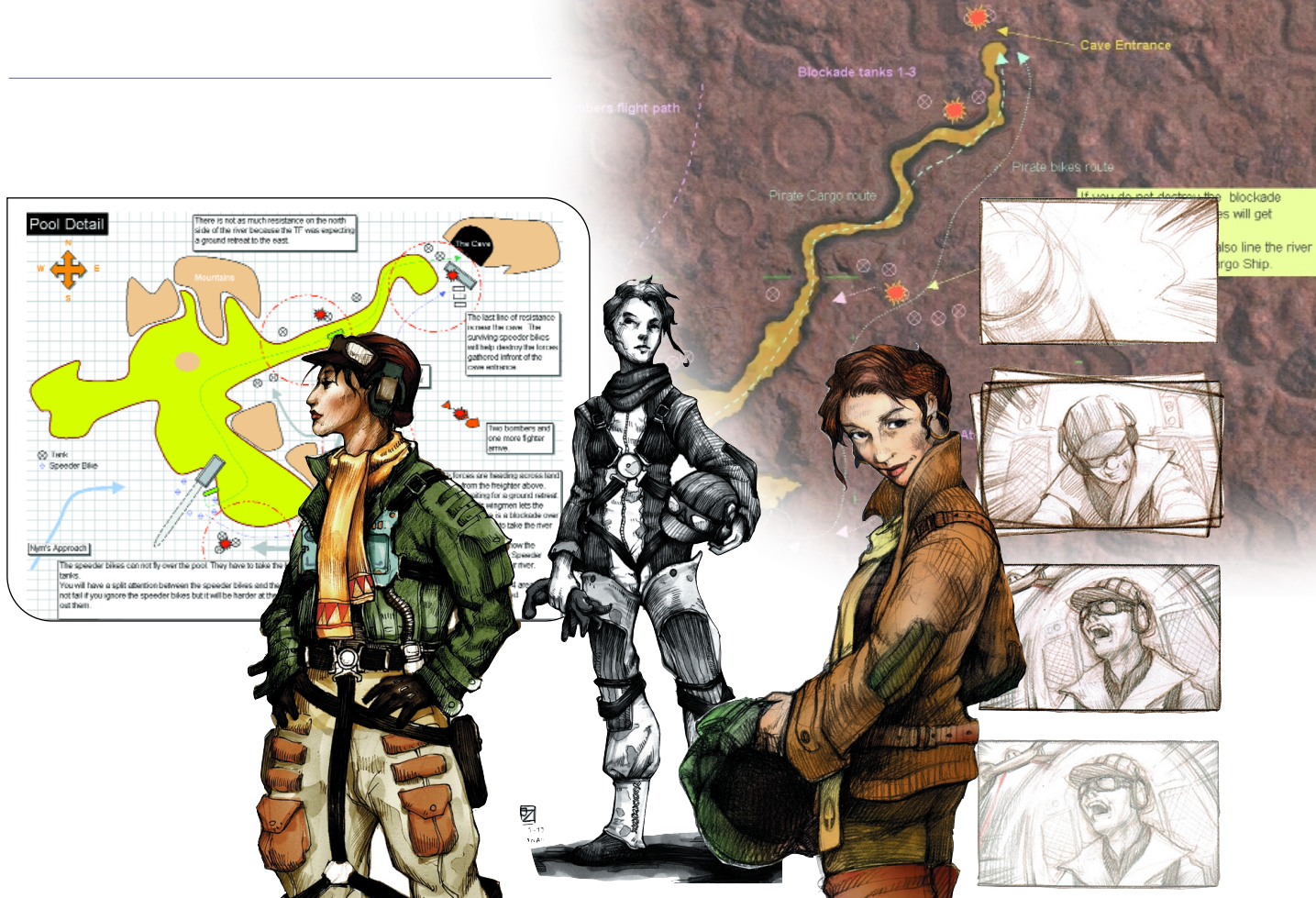
What Went Right

1. Good team communication. I've read many *Game Developer* Postmortems that blamed failures on a lack of communication, so I'm particularly proud that we got this one right. From the beginning of the project, Daron worked hard to impress on the programmers that it was the level designers and artists that would ultimately secure the success of Europa. As I became fond of saying, programmers build the picture frame, but it's up to the

rest of the team to provide the most important part, the picture. To bring this to fruition, the programming team needed to understand as best we could the way the rest of the team worked. While Andrew worked with lead artist Jim Rice and our world-builders to understand their workflows, Brett Douville, our AI and mission programmer, filled a similar role with the level designers. Brett scheduled regular "LD Days" with each individual member of the level design staff. This gave each designer the opportunity to meet with Brett on a regular basis and show him the specific challenges and problems that they were tackling in their missions. Europa periodically had full-blown team meetings where we could get together and kibitz about the overall state of the project. However, the most valuable meetings were at the subteam level. Both the programming team and the art teams would meet weekly to discuss the issues of the day, and each of these meetings would have an attendee from the other camp — a role we referred to as the "exchange student." This meant that if questions came up in the art meeting, for example, that required answers or input from a programmer, there would always be someone present that could give an informed opinion. Likewise,

as programmers would discuss issues or new features in their weekly meeting, the art or level design representative would be able to disseminate this information among the other team members. Finally, we relied on an internally maintained web site as a pivotal communications tool. We tried to make the site as comprehensive as possible, organizing areas along the lines of programming, art, level design, project management, and so on. When artists had questions about how to implement a particular effect, or a level designer needed a refresher on our class-file script syntax, there was usually a web page that they could be directed to that would answer many of their questions.

2. Project discipline. STAR WARS STARFIGHTER was a well-organized project. In the heat of battle it's all too easy to let requirement lists and schedules get lost in the shuffle of the moment. We were determined not to let this happen. As soon as our technology began to take shape we started to follow an iterative process of milestone planning and execution. These milestones were typically four to six weeks in duration, with no milestone extending longer than eight weeks. Milestones were also required to demonstrate some visual or gameplay aspect of the game. As a conse-



TOP LEFT & TOP RIGHT. These two images depict the design schematics for one of the Lok missions. Level designers made elaborate plans such as these for every level that appears in the game. BOTTOM CENTER. Early concept art for the mercenary Vana. FAR RIGHT. Storyboard depicting Rhys Dallow's ship getting hit.

quence, we had very few milestone tasks that looked like “complete the Foo-bar class”; instead we would have a milestone task that might read “Explosion smoke trails,” and the assigned programmer would know that completing the Foo-bar class was an implied requirement. By keeping our attention focused on a discrete and relatively small body of work, we were able to avoid the cumulative errors that invariably creep into longer schedules, while still allowing for demonstrable progress.

Most of the milestones were driven by the progress of the technical team. Programmers were solely responsible for estimating the duration of their tasks. We would occasionally adjust these estimates outward but would never change an estimate to be shorter. Tasks were structured so that the shortest scheduled task was never shorter than a half-day. Even if a programmer was certain that a task could be completed in less than half a day, experience clearly showed that the time would be lost elsewhere. Using these simple rules of thumb, we were consistently able to build schedules that were fair and accurate. Out of eight scheduled milestones, we never missed one by more than a handful of days. Best of all, most team members

completely avoided extended periods of crunch time. Like most game teams as they approach their ship date, everyone was working hard and often into the evenings; however, this period of time was short, and we never had to resort to all-nighters.

We also closely managed the process we used to distribute new binaries to the team at large. Since most of our development occurred on the PC even after making the decision to ship on the PS2, it was important that team members have timely access to stable builds of the game. We accomplished this through weekly public builds. Once a week we would package and distribute the current code as a full-blown InstallShield-compiled install. This provided team members with debug and production versions of the game, along with level design tool and art exporter updates. Predicting that public builds would become critically important, we tried to be as ruthless as possible about maintaining the build schedule. As we got closer to our ship date, the frequency of these public builds increased until we were performing new builds as often as three times a week. By this point we had a full-time staff member dedicated to managing the public build process and ensur-

ing that the distributed code met quality and functionality expectations.

3 • A well-executed PC-to-PS2 transition. Making the decision to move the project to the PS2 could have been a complete disaster. Yet, despite paying little attention to portability during the earliest stages of the project, the Europa code base was well positioned to make the jump to the PS2 platform. With the aid of strong and generally stable development tools provided by Metrowerks, the core port went off without a hitch. The biggest trick was on the graphics side, because this was clearly where we were most vulnerable. None of our programmers had any console experience, and none of us was up to the task of tackling the PS2's infamous low-level vector units. Enter LECgl.

LECgl was the brainchild of Eric Johnston and Mark Blattel, two of LucasArts' most senior console programmers. They had recently shipped STAR WARS: EPISODE I RACER for the N64, and they welcomed the opportunity to tackle a problem temporarily that was one step removed from the day-to-day pressures of a project team. Although Europa was the most immediate recipient of their efforts, Eric and Mark



LEFT. Several of the ship models developed for STAR WARS STARFIGHTER. TOP RIGHT. An early screenshot of the Havoc pirate bomber being chased by a Naboo Starfighter. BOTTOM RIGHT. Screenshot featuring the user interface created with Macromedia Flash.

were never officially on the project. Instead they worked in a support role, providing us with regular LECgl library drops and immediate “on-call” PS2 graphics support.

There was another, more subtle problem that we had to conquer when we made the decision to adopt the PS2 as our primary platform. Most of the team members were big PC game players, but very few of us played console games. Intellectually, we knew that there were huge philosophical differences in game design between consoles and the PC. Much of our original game design had used the X-WING games as a conceptual leaping-off point, wandering into the arcade action of ROGUE SQUADRON only when it suited us. Now that we were on the PS2 we recognized that our design priorities needed to be completely flipped. Instead we would use ROGUE as our primary point of reference and work from there, layering on game-play elements borrowed from X-WING as needed. As such, I think the final game demonstrates our successful indoctrination into the console mindset. We were having so much fun blowing things up that we had little desire to start adding simlike features to the gameplay experience.

4 • Macromedia Flash. As we approached the end of summer in

2000, we realized that we had a serious problem on our hands. Despite our best efforts, we still had not addressed the issue of our out-of-game user interface. We had a 2D virtual-page system that we were using for our HUD (heads-up display) symbology, and we had always planned to evolve that into something that could be used for what we called the “administrative interface.” However, in August, with the quality assurance department nipping at our heels and our ship date looming ominously in the distance, things were not looking good.

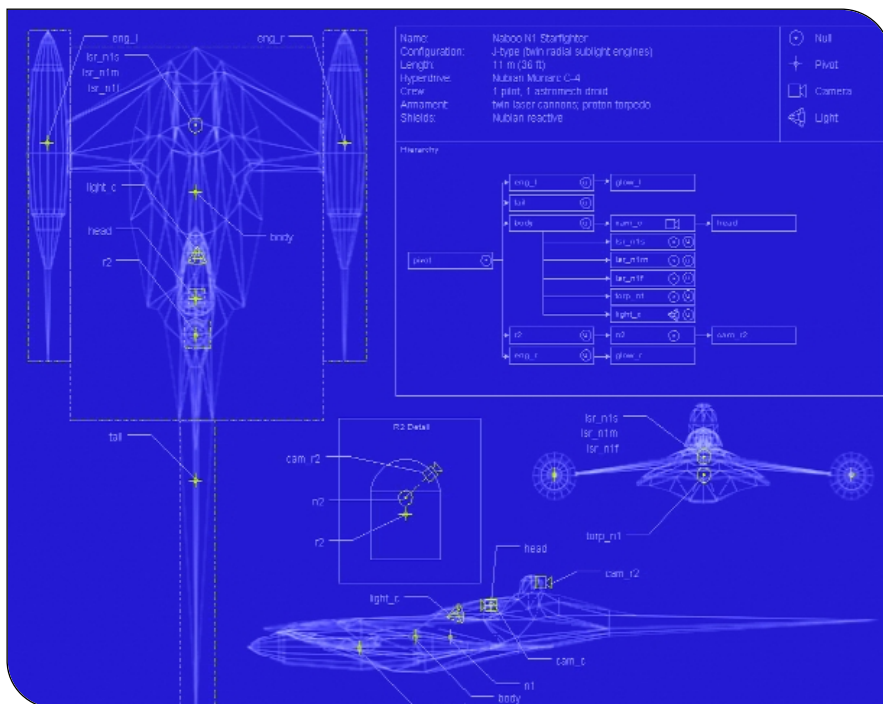
We had heard that a small San Francisco-based company named Secret Level was adapting Macromedia’s Flash technology for use in PS2 games. After meeting with company representatives, we were excited by the prospect. The Macromedia content-authoring tools were far more elaborate than anything we could come up with in the same time frame. We also suspected that there was a wealth of Flash authoring expertise available from out-of-house contractors which would help us smooth out the work load. Most importantly, we were very impressed by the intelligence and games savvy of the Secret Level staff. When we realized that building our user interface in Flash would significantly ease our localization efforts, we

decided to take the plunge.

Soon afterward we hired a design firm named Orange Design to help us implement our administrative interface in Flash. Orange not only had a ton of experience with Flash, but they also brought a technical perspective to the table. We knew that this technical emphasis would be critical for working with our programming team on integration issues.

Integrating Flash into the Europa engine was not a completely smooth process, however. Performance in the first-generation Flash Player was poor (current generations of the Player are now much faster), and we had to spend a lot of time integrating the user interface Flash movie with the core game systems. That said, the five months that a single half-time programmer spent on this task ended up yielding a user interface that was far beyond what we would have been able to custom-code in the same period of time.

5 • Good debugging systems. Our programmers built several tools that greatly helped our pursuit of high-quality code. One of the most instrumental was a Windows-only library that provided detailed stack-tracing information. This library was largely based on the code and concepts covered by John Robbins’



LEFT. A schematic for the Naboo Starfighter — one of the only elements in the game that was present in both the original design concept and the final product. TOP RIGHT. A Naboo Starfighter cruising over an early take on the rolling hills of Naboo. BOTTOM RIGHT. An early version of the Naboo Starfighter passing in front of a nebula in deep space.

“Bugslayer” column published in the *Microsoft Systems Journal*.

As is standard practice on many games, we built a custom memory manager that could detect when the application was leaking memory. However, unlike most implementations, when our memory manager detected a leak it could provide a comprehensive stack trace of arbitrary depth, leading directly to the leaking code statement. This capability represented a significant advantage over other implementations that could only provide the immediate location of the allocation request. If the memory allocation was being made by the Standard Template Library (STL) or one of our widely used utility classes, it was usually not enough to know what part of the STL or which one of our utility classes was the culprit. What we really needed to know was what class called the STL method that caused the leak. In fact, the leak was usually several steps up the call chain. Our stack-tracing library made finding these cases almost trivial.

We also incorporated stack tracing into our exception- and assert-handling systems. When the game encountered a hard crash, we trapped the exception and generated a complete stack trace; a similar process occurred when our code asserted. This information was initially reported

back to the user in dialog form. However, we also packaged up this same data and had the game send the programmers an e-mail detailing exactly where the problem occurred. This ended up being an invaluable tool for us. As a matter of practice, the Europa programmers got into the habit of checking the assert mailbox regularly. In addition to appraising the current stability of the code, we could also use this data to spot trends and note when people weren't being diligent about installing new builds.

In the end, we had an exceptionally smooth QA process because the bugs we did have were generally easy to track down and fix. There were no last-minute “heart attack” bugs that required us to set up camp and track a single problem for hours or days at a time. This made life easier on the programmers, but it also made things easier for the testing team and improved morale across the entire project.

What Went Wrong

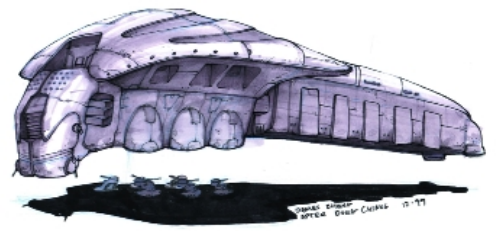
1. Staffing. As you can probably tell by now, staffing was easily the biggest problem the project encountered. Try as we did to manage staff retention, the team experienced an alarming amount of turnover, both in the programming and

art departments. This invariably made life harder for the people left behind, because the amount of work remained constant, but team members could not be replenished as quickly as they were lost.

This also meant that many of the team's junior staff members missed out on valuable mentoring or experienced spotty supervision by their leads. On the programming team, senior programmers were so busy that we had little time to train new team members. This led to a stressful sink-or-swim mentality which was difficult for new hires. Even relatively simple quality-control procedures such as code reviews were never instituted, since every moment of every day was dedicated to making forward progress on the game.

The staffing issue continued to dog us throughout the project. Even after we had regained some momentum, we still ended up losing two programmers and a handful of artists, all to the same online gaming startup. Although nine programmers contributed to the main code base at one point or another, the vast majority of code was written by the core group of four programmers who stayed with the project to completion.

2. Initial lack of detailed design. Europa was always envisioned as



having some sort of *Star Wars: Episode I* tie-in. During much of 1998, however, it was difficult to predict to what degree Lucas Licensing would allow this to happen. One of the barriers we encountered was the intense veil of secrecy that surrounded any Lucas-owned company involved with the movie property. Some of us had access to the script and the occasional rough-cut screening, but particularly during the first half of 1998 it was virtually impossible to learn the important details about the film needed to build a solid franchise title.

Initially we had assumed that the game should stay as far away as possible from the events of the film. Because we were going to be telling one of the first original stories set in the time line of the new film, we had no feeling for where the boundaries were with respect to planets, characters, vehicles, and the like. We were intimidated by the pervasive atmosphere of secrecy and general sensitivity of the *Episode I* story lines; the first game designs described a pirate war far divorced from the events of the film. In fact, the Naboo Starfighter was one of the only elements that could be found in both the first design and the film. As this design started to circulate, however, Licensing contacted the team and explained that the design contained too many pirate elements; they wanted the game to contain more elements from the film. The “moving target” nature of this exchange ended up being very disruptive and effectively paralyzed the design effort for weeks at a time as we wandered from idea to idea, wondering what fit into continuity with the film and what was straying into areas that we should keep away from.

The Europa team also had some pretty big shoes to fill. It didn't take long for us to realize that whatever we did was going to be directly compared to Larry Holland's previous X-WING titles. The Totally Games guys had been making games like for this for the better part of a decade, and they had gotten very, very good at it. Game players could rely on Larry to produce large, sophisticated games with well-designed features and compelling gameplay. This success had, in turn, incubated a dedicated and enthusiastic fan base that we knew would mercilessly scrutinize STAR WARS STARFIGHTER. Frankly, we were in a

no-win situation: if we deviated too far from the Totally Games designs, we risked disenfranchising some of our most loyal fans, but we also didn't want simply to copy Larry's last game either. Fortunately, once we decided to ship on a console, the design shackles fell away and we were free to chart our own path. While we realized that the hardcore X-WING players might not appreciate STAR WARS STARFIGHTER as much as the Larry Holland games, they were no longer our primary audience.

3 • Naïve approach to memory usage.

As quickly as we were able to get Europa up and running on the PS2, it took the programming team much longer to fully embrace the creed of the console programmer. Since Europa was originally intended to be a PC title and our programmers only had PC experience, it's not surprising that most of the code suffered from a bad case of “PC-itis.” I use this term to refer to programming practices that, while potentially portable to a console, are definitely not console-friendly. Our approach to memory allocation is a perfect case in point.

For starters, we relied on the STL for all of our container classes. On one hand we benefited from a bug-free and robust set of standardized collection classes. As an integral part of the C++ Standard Library, the STL contains a powerful toolset for general application development. We're big fans of the STL, and for the most part we can't imagine working on a project that doesn't use it. Unfortunately, depending on what containers you decide to use, the STL is notorious for making many small memory allocations. Our STL container usage was paralleled by our use of an uncomfortably large number of ANSI string objects. The ANSI string class is a great little class that makes dealing with character strings much easier than it used to be when we were all writing code in C. Like most STL containers, however, excessive use of the string class also leads to large numbers of small memory allocations. By the time we decided to port to the PS2, most of the damage had already been done.

As I mentioned earlier, our global memory manager's original focus had been memory-leak tracking, but now we needed it to help with our STL problem. We accom-

plished this by introducing the concept of bins, which were really just a hierarchy of fixed-length memory allocators. When the memory manager received a small memory request, it could very quickly and efficiently satisfy the allocation if the size of the request fell into the range serviced by our bins. We ultimately relied on the bins for both rapid memory allocation services and fragmentation management.

I should also note that we had a pretty rough time with memory fragmentation. Going into the PS2 port we suspected that fragmentation was going to be a problem. On the PC we had made an effort to generally clean up after ourselves in ways that would help reduce fragmentation, but we never made a concentrated effort to eradicate it completely, because we knew that in a pinch we could always rely on the PC's virtual memory system. One of my jobs during the last six weeks of the project was to build debugging systems that would give us detailed memory maps and then track down each fragmenting memory allocation one at a time. It was every bit as unpleasant as it sounds, and I urge those PC developers making the switch to consoles to take this lesson to heart.

4 • Not enough attention paid to performance.

There is little question that in the rush to implement features and ship the game on time, performance suffered. Part of this was due to having an inexperienced staff, and part of this was due to the fact that we had ported a PC code base to the PS2, but in truth most of us were so preoccupied with one issue or another that we had little time to revisit code with an eye toward optimization.

There was a pervasive attitude among many of us that we could safely ignore code problems until they showed up as hotspots on a profiling run. There is some merit to this strategy, since premature optimization efforts can be more wasteful than not fixing the code at all. But since profiling can turn up hidden problems in areas of the code that the team had previously thought complete or issue-free, it's important to start profiling much earlier than we did. For example, we had severe performance problems in our collision detection systems that we would have identified immediately if we had profiled sooner. As



it happened, by the time we realized that collision detection was working poorly, the best we could do was apply spot fixes instead of the large-scale reworking that the problem actually demanded.

Even after we started a fairly regular regimen of profiling late in the development cycle, we still didn't do enough of it. In the end, only one programmer did all of our profiling, and he was responsible for making the rounds and pointing out problems to other members of the programming staff. This was a real shame, because the Metrowerks PS2 profiler is a very nice tool, and most members of the team had uninstalled licenses. I should have made our developers responsible for profiling their own code and doing so at a much earlier stage.

5. Space-to-planet. If there was anything about the original STAR WARS STARFIGHTER pitch that met with widespread enthusiasm, it was the idea of seamlessly transitioning from planet-side environments to the depths of space and back again. Dogfighting close to the planet surface certainly has its own appeal, but there is something about the promise of being able to pull back on the stick and blast off all the way into space that is simply very, very cool. This high concept was so exciting to the team that the original game pitch featured this idea predominantly. In fact, in many ways this single feature was to define the game.

Well, it's a bit of a trick to actually pull off. First, there were the technical considerations. A planet is big. I mean really, really big. Even a small planet would require dynamically creating thousands of terrain tiles. Although most of these tiles could be procedurally generated, they would still need to be created and discarded on the fly; depending on the player's location, custom mission-area tiles would have to be streamed in from the hard disk, all while maintaining frame rate. Of course, since we wanted to allow the player to fly absolutely anywhere on the planet, ordering this data on the disk in a streaming-friendly format was problematic. We exacerbated the situation by requiring even our lowest-resolution terrain height maps to be much higher resolution than they really needed to be. This in turn

made higher theoretical demands on the streaming and resource systems.

This single feature had introduced a tremendous amount of technical risk to the project, and yet we had blindly charged ahead anyway because of the idea's inherent coolness factor. The technical issues, however, did not describe the full extent of our problems with this feature. Quite quickly we also came to realize that there were plenty of game design issues implied by the space-to-planet concept. For example, there was the constant issue of player craft speed. We felt pretty sure that our ships should have a top speed of about 450 miles per hour, because dogfighting and bombing ground targets becomes extremely difficult if you move much faster. However, at that speed it would take the player 20 minutes to achieve a low-planet orbit. To circumnavigate a small planet the size of the moon could take as long as 16 hours. Although we were able to brainstorm several fanciful solutions to this problem, most were time- or cost-prohibitive, and all of

our solutions threatened to shatter the illusion that you were in a small fighter craft, engaged in small, intimate battles.

Back to Earth

STAR WARS STARFIGHTER finally shipped in February 2001. While it was a little bit later than we had initially hoped, we burned our first set of master disks in mid-January, within three days of the "realistic" schedule projection that Daron had made a year earlier. While it certainly has its flaws, STAR WARS STARFIGHTER represents the culmination of an effort that involved almost 50 people, and it is a product that we are all very proud of. The lessons learned over the last few years, both positive and negative, are already starting to be used by other LucasArts teams, ensuring that the project's legacy will be with us long after the last copy of the game has been sold. *EF*

Games for Girls

Going . . .
Going . . .
Gone?

The subject of games for girls and women has always been a lightning rod for controversy. Some see it as positive outreach to an untapped 51 percent of the market; others see it as overzealous feminists looking for a cause to rally behind. No matter your point of view, the games for girls issue continues to dog the game industry and likely will for years to come. For several years I have moderated roundtables at the Game Developers Conference on whether we need to make games specifically for girls and women. This year however, we focused on a different aspect of the issue: Is the genre dead?

According to the census bureau, roughly 51 percent of the population in this country is female. A 2000 PC Data study found that 50.4 percent of online game players are women, while the Interactive Digital Software Association recently reported that females comprise 43 percent of game players overall. With so many girls and women playing games, why is there suddenly the perception that the games for girls genre is near-dead or dying?

Part of the answer may lie in the missteps that the industry has suffered in this arena, most notably the death of former media darling Purple Moon and publishers' struggles to gain market share even with strongly licensed properties such as Her Interactive's NANCY DREW series. Why is the target audience not responding well to products designed specifically for their gender? There are three key reasons.

The fun factor. One thing far too many publishers overlook in designing "female-friendly" content starts at the base level: good game design. The road to the discount software bin is often paved with the good intentions of game developers and publishers seeking to portray females in a positive light. Instead of a fun, engaging game experience, we get self-esteem in a box — uplifting messages of self-empowerment telling us that girls can do anything, be anything, and go anywhere.

There have been a variety of studies showing the differences in the way men and women play games. One notable difference is the way we tackle obstacles. Confront a female player with an obstacle and she'll analyze it, discuss it, and then try to work out the best solution possible. Confront a male player with the same



Illustration by Claudia Newell

obstacle and he'll want to break it down as quickly as he can and then move on to the next challenge.

We don't have to whack girls and women over the head with cheery, upbeat self-esteem messages. Instead, developers need to take a look at

the differences in how we play games, then incorporate them into their game designs so that both genders enjoy playing. As one person during the roundtable suggested, "We need to make games for people, not games for boys and games for girls."

It's a man's world. The fact is that the game industry is still very much a boys' club. This is an industry that was founded by males for males, and in the beginning women were limited to traditional women's roles such as human resources, public relations, and the ever-present booth babe. While women are making strides in programming, game design, project management, and marketing, the glass ceiling has yet to be shattered completely.

Interestingly, one of the laments I heard repeatedly from male attendees at GDC was that they couldn't find female game designers, even when they desperately wanted to hire them. Most wanted to add women designers to their teams to broaden their game's appeal to women but, sadly, were unable to find enough applicants to fill even one or two positions.

We as an industry need to do more to encourage women to take jobs as game designers, programmers, and producers. The more we can incorporate a woman's point of view into our games, the more likely we are to attract women players. As Will Wright, the creator of THE SIMS, noted, "I don't think there's a magic game design formula to appeal to women. The only way it's going to turn around is when you get more women in game design."

Software stores: enter at your own risk. One of the factors that is highly overlooked when it comes to attracting female players is the way games are presented by publishers, retailers, and the game

continued on page 63

continued from page 64

press. I recently conducted a nonscientific poll of my women friends on this issue. Unfortunately, what I found was not surprising; here is a sampling of some of their comments:

“I hate going into software stores. If I go to Electronics Boutique, I’m almost always the only woman in the store unless it’s somebody’s mother looking for her kid.”

“Software stores creep me out.”

“I am so tired of the ads in the game magazines. It’s either chicks with huge chests and a big gun or junior high school toilet humor. It’s like some kind of cut-down version of *Playboy*.”

The way we market and advertise our games is key in the female market. Product packaging and in-store advertising are targeted at the male populace rather than being gender-neutral — highly rendered, hypersexualized images of women are very common on box fronts and collateral such as posters and standees. Game magazines are notorious for ads filled with vio-

lence, gratuitous sexual innuendo, and puerile humor.

Bearing in mind that a recent IDSA study showed that women are responsible for nearly half of the games purchased in the U.S. today, how can we as an industry expect girls and women to feel included when most of the tools we use to sell our games cater to men? Software stores and game magazines remain hostile territory for many women, effectively locking them out as part of the game-buying public. Until publishers, retailers, and the media rethink how they present games to women, women will continue to take their considerable spending power elsewhere.

Every year I expect to be giving my “games for girls” roundtable for the last time, thinking that people must be tired of hearing me beat the same old drum. However, I have realized that until the industry matures into one that openly welcomes game developers and players of both genders, this will remain a hot-button issue. As much as I love speaking at GDC, I still hope that next year is the last year we have to talk about this subject. 🍷

MELISSA FARMER | *Melissa is a longtime veteran of the game industry and has worked for companies including Stormfront Studios, Titus Games, and TalonSoft Inc. Melissa is a former writer for GrrlGamer magazine and served as the executive director for the Computer Game Developers Association. She is currently the product marketing manager for UNREAL TOURNAMENT at Infogrames Inc.*
