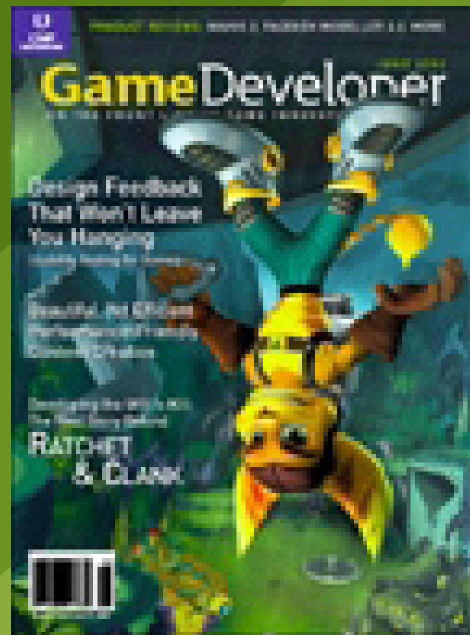
gd

GAME DEVELOPER MAGAZINE

JUNE 2003



UBM
Tech

GAME PLAN

LETTER FROM THE EDITOR

Credits and Debts

In an industry where days are long but memories are often short, game credits are vital, but they sometimes pose problems for developers and employers. If you've never had to confront the issue head-on in your professional career, consider yourself lucky. When a former employer owns every last byte of your work on a game, your name in the credits may be all you have to show for your work and help land your next job. There are opportunities for abuse on the parts of both employer (denial of credit) and employee (credit inflation or misrepresentation); fortunately, there are incentives for both sides to resolve the issue.

One stumbling block is the lack of industry-wide standardization of job titles and their respective scope of responsibility, which creates opportunity for abuse from both employers and employees. In Hollywood, people work on contracts and deal with unions that can specify what a "best boy" is and does, so it's hard for him to show up on his next project claiming he was a second-unit director. On the flip side, the union would be able to protect him through arbitration if he fulfilled the terms of his contract but the film's fun-loving producers decided to credit him as "goat boy" instead.

More significant than the mere lack of job-title standardization in the game industry is the anarchy that characterizes the methodology around which credit files are typically created, maintained, and implemented at the end of a project. Many employers still have a make-it-up-as-they-go-along approach to generating game credits, both because there is no industry-standard methodology to follow and because disputes often arise after the fact and are forgotten by the time the next title ships. The IGDA is looking at creating a voluntary set of standards for companies to follow, which would be of great help to employers and great relief to developers.

The use of short-term contractors and mid-project turnover of full-time employees represent two big sticking points in credit determination. As lengths of game projects swell from "Gee, I could have gotten a master's instead" to "I gave you the best years of my life!" the chances of an entire team remaining intact from start to finish are increasingly slim. Since most work in the game industry is done by studio employees rather than contractors, mid-project departures wreak havoc with credit claims. One company might punish a lead who left a month before ship by eliminating him or her from the credits, while another might include a low-level person who got fired for incompetence after a few months. It's a crapshoot for hardworking developers who don't know whether they are choosing between their next opportunity and credit for their current work.

What can you do to protect yourself? First, make it a priority to protect the investment your hard work on a project represents. This means that you can't rely on your employer's good intentions. Raise the subject of credits with the project or studio manager when you are hired, or each time you start on a new project. Find out who administers the credit file and how it is reviewed. There should be a consistent framework to determining criteria for inclusion (and exclusion), and you should get it in writing. Such a framework benefits employees by managing their expectations and employers by giving them the ability to justify and defend their final decisions.

Game developers are not by nature a vainglorious lot; most are still doing it out of love for the work rather than a shot at notoriety. The idea of pushing for verifiable crediting processes at your company may even feel unseemly or selfish. But credits are an area where there should never be surprises. It's worth it to both you and your employer to establish what your company's crediting standards are.

Jennifer

Jennifer Olsen
Editor-In-Chief

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

Publisher
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL

Editor-In-Chief
Jennifer Olsen *jolsen@cmp.com*
Managing Editor
Everard Strong *estrong@cmp.com*
Production Editor
Olga Zundel *ozundel@cmp.com*
Art Director
Audrey Welch *awelch@cmp.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Jonathan Blow *jon@number-none.com*
Hayden Duvall *haydend@3drealms.com*
Noah Falstein *noah@theinspiracy.com*
Advisory Board
Hal Barwood LucasArts
Ellen Guon Beeman Monolith
Andy Gavin Naughty Dog
Joby Otero Luxoflux
Dave Pottinger Ensemble Studios
George Sanger Big Fat Inc.
Harvey Smith Ion Storm
Paul Steed Microsoft

ADVERTISING SALES

Director of Sales/Associate Publisher
Michele Sweeney e: *msweeney@cmp.com* t: 415.947.6217
Senior Account Manager, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 828.350.9392
Account Manager, Northern California & Southeast
Susan Kirby e: *skirby@cmp.com* t: 415.947.6226
Account Manager, Recruitment
Raelene Maiben e: *rmaiben@cmp.com* t: 415.947.6225
Account Manager, Western Region & Asia
Craig Perreault e: *cperreault@cmp.com* t: 415.947.6223
Account Representative
Aaron Murawski e: *amurawski@cmp.com* t: 415.947.6227

ADVERTISING PRODUCTION

Vice President, Manufacturing Bill Amstutz
Advertising Production Coordinator Kevin Chanel
Reprints Terry Wilmot e: *twilmot@cmp.com* t: 516.562.7081

GAMA NETWORK MARKETING

Director of Marketing Greg Kerwin
Senior MarCom Manager Jennifer McLean
Marketing Coordinator Scott Lyon

CIRCULATION

Game Developer is BPA approved

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

SUBSCRIPTION SERVICES

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

INTERNATIONAL LICENSING INFORMATION

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

CMP MEDIA MANAGEMENT

President & CEO Gary Marshall
Executive Vice President & CFO John Day
Chief Operating Officer Steve Weitzner
Chief Information Officer Mike Mikos
President, Technology Solutions Group Robert Faletra
President, Healthcare Group Vicki Masseria
President, Electronics Group Jeff Patterson
Senior Vice President, Global Sales & Marketing Bill Howard
Senior Vice President, HR & Communications Leah Landro
Vice President & General Counsel Sandra Grayson
Vice President, Creative Technologies Philip Chapnick

GamaNetwork

WWW.GAMANETWORK.COM

INDUSTRY WATCH

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

Nintendo cuts royalty rates. Nintendo Co. Ltd. has lowered its royalty rates that third-party publishers must pay the company. The company hopes this move will help lure more titles for the Gamecube at a time when Nintendo lowered its current fiscal-year outlook. The company expects net profit for its current fiscal year to be approximately $548 million, lower than the initial estimates of $665 million. The company attributed low Gamecube sales for the lower profits.

Mythic infusion. Mythic Entertainment, developers of the MMORPG DARK AGE OF CAMELOT, has received a $32 million investment from TA Associates, a private equity and buyout firm. The company plans on using the cash to expand their online player base and create new titles.

Interplay's full-year results down. According to year-end reports, Interplay's 2002 revenue was $44 million, a 22 percent drop from 2001's revenues of $56.4 million. However, the company ended the year with a net income of $15.1 million, compared to the previous year's $46.3 million net loss. The company attributed a $28 million gain to its sale of Shiny Entertainment to Infogrames.

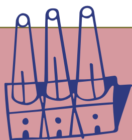
Mythic Entertainment, developers of DARK AGE OF CAMELOT, received a $32 million infusion.

Acclaim saved from within? Acclaim CEO Greg Fischbach and senior executive vice president James Scoroposki fronted $1 million each of their own money in hopes of keeping the publisher from being delisted from the NASDAQ due to low share price. In exchange for the cash infusion, the two executives were each issued 2 million shares of company stock, plus other warrants.

ATI revenues up, but profits down. Blaming a decline in royalty income, ATI Technologies' second-quarter earnings dipped slightly to $318.5 million from last year's $322 million. Charges in the quarter included an $8 million class-action lawsuit settlement, $2.8 million in the closing of a European manufacturing operation, and other expenditures.

Xbox cuts European prices. In a move calculated to position itself ahead of Nintendo, Microsoft lowered the price of its Xbox console in the European market. The move, Microsoft's third European price cut in less than a year, makes the console cheaper than the PS2 and about the same price as the Gamecube.

Send news items and product releases to news@gdmag.com.



THE TOOLBOX

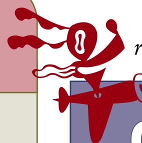
DEVELOPMENT SOFTWARE, HARDWARE, AND OTHER STUFF

Two 3DS Max plug-ins from Turbo Squid. Turbo Squid has released AfterBurn 3 and Kaydara's HumanIK plug-ins for Discreet's 3DS Max. AfterBurn 3 enables rendering of realistic effects ranging from clouds, dust, and explosions to liquid metals. HumanIK's features include automatic character rigging, a BodyGenerator Max script, and other features. www.turbosquid.com

Maya 5 unveiled. Alias|Wavefront announced the newest version of their 3D modeling and animation program, Maya 5. The new version features four rendering options — the Maya software renderer, Mental Ray for Maya, a new vector renderer, and a hardware renderer — plus enhanced character animation tools, new modeling tools, API updates, new data exchange options, and expanded polygon mesh-editing tools. www.aliaswavefront.com

New game editor development tool announced. Adventurerland Entertainment has announced its Lab Technology Construction Kit, software that includes a built-in compiler/linker that creates and deploys a game engine's output. The kit supports DirectX, OpenGL, and SDL. www.adventurerland.com

Softimage reveals Softimage|XSI 3.5. Softimage announced the newest version of their modeling and animation environment. Version 3.5 features automatic symmetrizing of polygons, flattening of UVs, more realistic hair generation, and UI support for Cg and DirectX vertex and pixel shaders. www.softimage.com



UPCOMING EVENTS CALENDAR

CHRISTIAN GAME DEVELOPERS CONFERENCE
CASCADE COLLEGE
Portland, Ore.
July 25–27, 2003
Cost: TBA
http://cgdc.graceworksinteractive.com

SIGGRAPH 2003
SAN DIEGO CONVENTION CENTER
San Diego, Calif.
July 27–31, 2003
Cost: $50–$950
www.siggraph.org/s3003

CLASSIC GAMING EXPO
JACKIE GAUGHAN'S PLAZA HOTEL
Las Vegas, Nev.
August 9–10, 2003
Cost: $35 weekend pass
www.cgexpo.com

Havok 2: All Rag-Dolled Up

by justin lloyd

"Does it do anything but driving games?" is a question that draws groans and smiles simultaneously from the Havok team, makers of their eponymous game physics middleware that recently received a major upgrade to version 2.0.

In response to what must be an all-too-frequently asked question, Havok 2.0 broadens its scope to include character physics much more prominently over earlier versions. In that sense Havok 2.0 is not an upgrade, it's a whole new product. New demos show off this functionality to good effect over previous releases.

New and improved. In the original Havok we heard a lot about dynamic characters — rag doll — but it was more akin to having a dynamic corpse. Havok 2.0 extends what is possible, thereby losing a lot of the dead-body feel of its predecessor. I'm not an animator by any stretch but I was able to develop my own walk cycles and attach the Havok physics system and character control far easier than with earlier versions. Havok has realized that I need to dictate full control over character look and movement in the game, both player and nonplayer, because that's what the game player wants. I don't always require accurate physical simulation, for example when a more cinematic or fun-focused effect is desired. Havok attempts to provide this control, and while far from perfect it's a step in the right direction. I was able to emulate parts of the standard zombie demo easily enough that as my animation walks forward I "detached" the head, allowing it to loll freely. When hitting the character with an impulse, such as simulating a shotgun blast, I switched state to playing a "knocked back" animation, detached the arms and let the physics control the flailing arm movement and torso rotation. This could have been extended to letting arms dangle limply and legs drag heavily along the ground when targeted. I was also happy to see a proper section in the Havok manual for handling characters and control, so you aren't left floundering.

You may not have heard this, but Havok does driving games! There is obvious new functionality in the vehicular component of the SDK, but the important part for me was how easy it is to create vehicles now. The original Havok Vehicle SDK provided a lot of black-box functionality that was poorly documented and nonintuitive. With earlier releases, I spent a lot of time spinning my wheels, literally, trying to figure out how to balance all the forces. Havok now assumes that you don't inherently know what anti-roll bars are for, how to tune the suspension based on terrain and vehicle type, and how to calculate friction circles for cornering. Havok details specifically how their Vehicle SDK black box works and assumptions made, removing a lot of past guesswork from the equation.

Another thing that changed was automatic construction of object groups. Havok 2.0 introduces "Islands" so that you no longer have to take an educated guess as to which group in which to place your object so that the collision system is optimal. Islands are automatically constructed at run time, containing small groups of objects based on the simulation criteria. I had a previously existing prototype inside of which was a spiral of dominos that could be knocked over. Originally, I placed all of the dominos in their own group, which was fine, so long as the player didn't attempt to knock over multiple dominos at once, causing multiple cascades, as that would slow down the simulation. Havok 2.0 took care of the grouping for me; once a few dominos came to rest, they formed their own "simulation island" and became quickly deactivated. This relieves a huge burden from me having to take what was, at best, an educated guess as to which group objects should be placed in. I also no longer need to verify that someone else has placed objects in the wrong group.

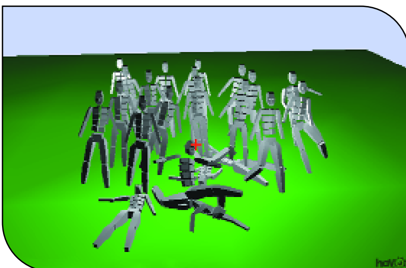
JUSTIN LLOYD | *Justin has over 18 years of commercial game programming experience on almost every released platform.*

HAVOK GAME DYNAMICS SDK 2.0

HAVOK
www.havok.com
Sales and Engineering:
San Francisco, Calif.
(415) 543-4620
Engineering:
Dublin, Ireland
+353 (1) 677-8705

Other platforms. I've only been able to work with Havok 2.0 on VC++ 6.0 running under Windows XP. Circumstance did not permit me to compile on Sony and Nintendo dev kits, so my only experience with Havok 2.0 on those platforms is playing with the interactive demos at GDC. From what I was able to glean from their new Visual Debugger, a module similar to the real-time VTune, the frame rates on complex scenes, such as the zombie demo with a dozen animating rag dolls, and character control were rock steady. CPU and memory usage never spiked above 50 percent, usually hovering around 20 percent, and when it did climb it was only for one or two frames when multiple characters attempted to interpenetrate. This was a pleasant surprise, as in the past with Havok 1.7 I could bring my 3GHz, 512MB, 64MB Radeon 9000 to its knees with a half-dozen rag dolls. Looking through the documentation and API headers reveals that Havok has gone to great lengths to ensure that each platform is properly targeted, the PS2 build making good use of the scratchpad and vector units.

How useful Havok's "Visual Debugger" will prove is questionable. It certainly gives you insight into the work being performed by the physics engine, but I'd like to see more hooks to allow integration into standard profiling tools such as VTune and ProDG's tools for Gamecube and PS2. I've worked on projects where the team has provided a profiling solution, and my experience with the Havok VD suggests that it does not provide the hooks that will be needed.



Screenshot from the author's second demo, blending animation and physics.

Support. Unlike a few middleware companies I have dealt with, Havok is going to great lengths to ensure that the people providing tech support for the licensees are qualified to do so, ex-game and physics programmers to a man. They're tech support is second to none. From prior experience with the 1.8 SDK on a real project, technical queries submitted late in the afternoon would elicit a response by the time I returned to my desk the next day. Still, Havok seems to understand that as a middleware provider, customers' project deadlines don't wait for them. Pertinent questions that would apply to other developers are often placed in the FAQ — even if they are only asked once — along with useful code snippets; even small features are quickly rolled back into the main Havok code base. I saw just how quickly this happens when attempting to create a vehicle that behaves as the Warthog in HALO, with all four wheels steerable. Within 18 hours I had my answer, and a day later it was a FAQ and code snippet on how to modify the Vehicle SDK. Havok supplies 95 percent of the source code once you move beyond evaluation, and that can save a lot of time when you are hunting down an elusive bug or attempting to optimize your game. Havok is quite happy to let you look inside their physics black box as much as you like.

Documentation. I'm of two minds concerning the way that Havok is pursuing documentation. The learning curve for the Havok SDK can be steep, and one reason is that they rely more on demos than documentation to illustrate a point. That causes two learning bottlenecks: First, you have to spelunk through many, many source files looking for the demo

THE TESTBED

Each project is different, and a physics simulation, like Internet traffic, changes moment to moment, making it difficult to provide hard data that you could form a judgment around. My test rig was 3GHz, 512MB DDR 2100, 64MB Radeon 9000–equipped notebook computer running Windows XP and DirectX 9.0. I assembled two demos for this review.

The first test project consisted of approximately 150 objects in close proximity, 10 of which were considered complex; small-scale versions of the St. Louis Arch that were pinned, i.e. infinite mass and immovable. The rest of the objects were mostly simple geometric shapes (boxes, wedges, and spheres) that the player could interact with in varied fashion. There were also a half dozen rag dolls consisting of 15 bones each; a rag doll was connected to its immediate neighbor via 2D dashpots, forming a loosely coupled chain. The player was represented as a single, two-wheeled vehicle that rode as a child's tricycle, constructed from one large wheel at the front, and a single small wheel positioned just slightly behind for stability. Due to the small distance between the wheels, the vehicle possessed a very small turning circle, enabling the player to negotiate the tight environment.

The second demo — a simultaneous blending of animation and physics — consisted of two dozen rag dolls animating a walk cycle that the player could shoot at with the mouse. The quoted memory footprint covers the entire demo,including graphics.

	Peak Frame Rate	Average Frame Rate	Peak Memory Usage	Average Memory Usage
Demo 1: Large, 150-Object Scene	120	80	10.1MB	9.4MB
Demo 2: Animation and Physics	174	98	2.9MB	2.7MB

YOUR MILEAGE MAY VARY: HAVOK 2 USERS WEIGH IN

"We've had a positive experience with Havok, although because we're about halfway through our current game life cycle, we still have issues that we need to resolve. If we are happy with Havok at the end of the game, I can certainly see us using it again, as much of the hard work of learning the API will already be done. The documentation could be better, though; a real boon would be to have more code examples in the docs themselves, rather than having to track down the usage inside a demo."

— Mark Baker, Mucky Foot

"Even with Havok it still takes significant work to combine gameplay and physics in a way that they enhance each other rather than detract from one another. When we originally looked at physics middleware a year or so ago, Havok looked like the most complete product on all platforms."

— Sam Baker, Paradox

"We aren't using the newest release yet, but during development we've given them lots of feedback and many of those issues have been addressed in Havok 2."

— Jay Stelly, Valve

or snippet of code that you think you need (the "I'll know it when I see it" approach). Second, because the Havok demos are built around a common framework of code, the body of which pulls in every feature that Havok provides, it requires more support functionality. You have to build up a mental roadmap of that in your head, learning what's important and what's immaterial to the particular demo before you can delve in to the small piece of code that you really should be concentrating on. The demos are good starting points, but better documentation that shows the bare minimum you need and why you need to do the particular steps, would make the job easier.

Terms. Licensing terms vary based on numerous factors such as the number of titles and platforms involved, and there are numerous support options from which to choose. One thing that is consistent is that license fees are one-time-only, with no royalties involved.

Final word. Havok is beginning to show its maturity, both as a middleware company and as a viable SDK for a new project or to replace your in-house solution. Version 2 shows great strides forward beyond Havok's driving-game roots, especially in the realm of character-based physics, making it worthy of evaluation by developers on a wide range of projects.

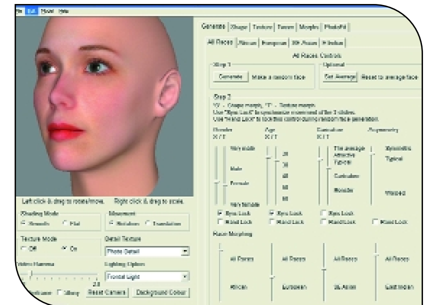
Singular Inversions' FaceGen Modeller 2.2

by michael dean

FaceGen Modeller 2.2 is the newest version of the face and head creation software from Singular Inversions. It has been designed to allow a user to create custom, unique faces faster than traditional 3D modeling packages typically allow.

Getting started is easy and fun. Upon launch, a default head loads into the shaded viewport, and I was immediately able to create random faces with just a push of a button. There is also the option to load faces that have been previously custom-created in the software.

To customize the premade and randomly generated heads, FaceGen comes complete with a simple yet powerful modeling toolkit. It differs from a traditional 3D modeling package in that geometry is not directly manipulated on the vertex/face level but through a series of sliders that control all aspects of your model. For example, if I create the face of a young woman and then want to change the model to reflect an older age, rather than push and pull vertices I use two sliders that control age, one for the geometry and one for the texture. Move them forward, and cheeks lose their fullness, the nose grows, and the skin weathers, all based upon the face's natural aging process. It works remarkably well, and this is how everything in FaceGen functions. There are also sliders that control masculinity and femininity, race,



A custom head created in just a few minutes from a few simple snapshots of my subject. No additional editing was needed, the default result was amazingly accurate.

symmetry, and realism.

Taking this to a deeper level, there are sliders that make up, and append, subsets of these more general categories. The user can go into a much more detailed slider group and fine-tune features like the character of a nose. Users can also adjust for a heavier brow or make a longer face, for example. The ability to easily create faces that differ greatly from one another doesn't disappoint. Morph targets can be generated using combinations of emotion sliders, as well as phoneme sliders.

FaceGen's built-in faces are customizable, and there's plenty to do with the templates and modification tools provided. However, many artists will want the ability to create heads from unique material, which is where the PhotoFit service comes into play. With PhotoFit, simple snapshots of people can be made into full models, all fully modifiable, as they are with the template faces. After the images are acquired, the wizard interface brings them into the software and the images are sent via Internet to Singular Inversions, who converts the photographs (one or two photos are supported) into a FaceGen file, which can be loaded into Modeller, edited, applied to a mesh, and exported as a model with UVs and texture. The results are amazing and look as if they came from a 3D scanner. This additional feature is fast, it took me about 20 minutes to receive custom-generated FaceGen files. PhotoFit also requires an additional investment, starting at $9.50 per face,

after an initial 10-face credit.

FaceGen exports to the native formats of most popular 3D packages. The models are a bit heavy on geometry, but upon import into your favorite 3D package, detail is easily reduced. I converted a 6,000-polygon FaceGen head into a very nice 550-polygon 3DS Max head by using only Max's built-in MultiRes and then flipping a few edges. The entire process of creating this head took 10 minutes, and the results were excellent.

Even if the software cannot create exactly the head you have envisioned (for example, getting the bulbous nose I envisioned for a character proved impossible just using FaceGen), rest assured that FaceGen will give you a very solid starting point. At $495, FaceGen is a robust package with a lot going for it, and an artist could easily expect to save hundreds of hours of time in the creation of unique faces for any given project.

★★★★ | FaceGen Modeller | Singular Inversions | www.singularinversions.com

Michael Dean is currently an artist at Ion Storm in Austin, Tex.

Real-Time Shader Programming by Ron Fosner

reviewed by jeremy jessup

In *Real-Time Shader Programming*, Ron Fosner describes the essential elements necessary for developing shaders in a very approachable full-color book that spans just over 400 pages. Your $49 also gets you a CD with a beta version of ATI's RenderMonkey and coded examples of many of the shaders discussed in the text.

Beginning with elementary vector math, the book moves quickly into lighting theory. The lighting chapter highlights the mathematical approximation of physically based lighting using the traditional ambient, specular, diffuse, and emissive colors in a scene. Representations for reflection and refraction are derived from Snell's law and Fresnel equations. Finally, non-photorealistic rendering (NPR) from cel shading, tonal art maps, and hatching is covered through pictures and a wealth of external references. The chapter makes for an enjoyable read by providing an understandable background to lighting techniques for non–seasoned graphics programmers.

Fosner describes how to set up the DirectX pipeline to use shaders. While he touches on some of the nuances you're likely to encounter, the DirectX section seemed a bit sparse compared to the earlier chapters. The DirectX setup calls specific to shaders were well documented; however, the chapter didn't dwell on creating the pipeline.

The book then describes several current shader creation and visualization tools. This chapter is relatively short, perhaps due in part to the newness and hence volatility of cutting-edge shader tools. While high-level shader tools, such as Nvidia's Cg and Microsoft's High-Level Shading Language were briefly mentioned, the focus was on the shader language primitives. As such, it provided a sound fundamental shader approach that is universal to all higher-level shader implementations.

With the groundwork firmly in place, a wealth of shader examples follows. Starting with the minimal vertex shader, additional functionality is layered to build more complex shaders. Sample shaders are developed using the lighting equations presented earlier. While it may take a little time to digest some of the more sophisticated examples, such as the cartoon shader, the text provides adequate descriptive detail coupled with helpful color pictures to make it easier.

The final chapter provides a vertex and pixel command reference. Each command describes the supported shader version, usage, and a short example. The book covers shader implementations for both DirectX 8.x and DirectX 9. Differences between the two versions are noted throughout the sample code and reference section. When appropriate, additional notes on specific DirectX versions are also provided, and Fosner does a good job of providing references throughout the book for further information on a subject.

While having familiarity with the rendering pipeline, I found this book very approachable and easy to understand despite not being a low-level graphics programmer. The writing and companion tools provided challenged me to explore the world of shaders and attempt to write some of my own. The tools were a great aid, since they freed me from having to write my own engine and instead let me focus on the actual shader code. Writing in pseudo-assembly may not seem like fun, but it was — especially when I could experiment with one of the precoded routines Fosner supplied and view the results of a vertex or pixel shader routine through RenderMonkey instantly.

Shaders will play an increasingly important role in game development. Fosner's book presents the introductory groundwork necessary for developing custom shaders. For the experienced shader programmer the book's depth may not satisfy, but to those new to shaders or want to experiment with different rendering effects, this book is a great place to start.

★★★★ | *Real-Time Shader Programming* by Ron Fosner | Morgan Kaufmann | www.mkp.com

Jeremy Jessup is a programmer for

No One Markets Beer Forever
Monolith's Samantha Ryan

Before joining the game industry in 1996, Samantha Ryan had a solid 10-year career in broadcast marketing, working on projects for Infinity Broadcasting, Miller Brewing Company, and Frito-Lay. Ryan joined Monolith in 1998 and was recently promoted to president. Samantha's roles are varied, from pursuing new projects and partnerships to overseeing trademarks and legal agreements. "Conversely," she admits, "if the conference room is messy, I'll clean it up."

We caught up with Ryan to find out how she leads a company like Monolith, and how her marketing background comes into play when dealing with videogames.

After 10 years in broadcasting, Samantha Ryan decided it was time for a real career and joined the game industry. She is now Monolith's president.

Game Developer: How has your past experience in entertainment marketing crossed over into your work at Monolith? How does it affect the way you lead the company?

Samantha Ryan: It's true my past has been a little unusual, although that doesn't seem to be uncommon in this industry. For 10 years prior to entering the game industry I worked in broadcasting in both a production and a marketing capacity. The knowledge I picked up there has definitely shaped my approach to developing games. For example, there's something about the marketing process itself that is incredibly intriguing. I'm not talking about creating box art or ads, although those are certainly challenging. Rather, positioning and the psychological aspects of team dynamics as well as consumer marketing are fascinating subjects and are worth studying by anyone in senior management, regardless of duties or title.

GD: When developing a game idea — like NO ONE LIVES FOREVER — how important is the brand's marketability, in addition to its gameplay?

SR: A brand's marketability is an aspect of development that was de-emphasized in our early years. However, by this stage in our maturity as a developer, it's become very important. We work closely with our publishers on each title's overall positioning as well as the ongoing marketing and publicity efforts.

When developing a new intellectual property, we strive to create a robust universe populated with compelling characters. The more well-rounded your property, the better it will lend itself to application in other mediums. NO ONE LIVES FOREVER taught us a great deal, both things we did right and things we need to do better.

GD: With three teams working on different projects under one roof, how do you balance out projects, personnel, and other resources so that not everyone is in crunch mode all the time?

SR: Over the years we've learned to carefully select and schedule the timing of our projects; typically we prefer no project finish within six months of another project. At both the beginning and tail end of large projects, where you don't need a full production team, people tend to move around to help on other titles. This strategy allows us to keep our teams and personnel together for the bulk of a project, but also to give people some variety, and test them out on other teams.

GD: What has Monolith done to create an environment where its employees can have a sense of fun with their work and stay motivated while still being able to meet milestones?

SR: This is a difficult challenge for any management team. A great environment starts with great projects, so we're pretty selective. In addition, we plan company events, such as movie screenings, outdoor parties, and the like. We have also chosen an office-based seating plan rather than a cubicle-based arrangement; we find people enjoy having a bit more privacy, and that communication isn't hampered by it.

GD: How do you keep your creative teams creative when they are working on licensed content and sequel material?

SR: Staying creative on these types of projects has never been a problem for Monolith. It's all about the attitude with which you approach these efforts. In addition, we've had some wonderful properties with which to work: ALIENS VS. PREDATOR, TRON, and THE MATRIX.

GD: What is the key to retaining your best employees?

SR: The best projects. Certainly maintaining competitive benefits and salaries are important, but most Monolith employees, including myself, work in this industry because we truly love gaming. Therefore, finding great projects for our teams is probably one of our most important priorities. This is no easy task, but definitely worth the effort.

GD: You had most of the team from NOLF return for NOLF 2. How do you encourage a team to keep a fresh approach while building on a franchise?

SR: Because it's usually right at the end of a project that a game finally takes shape, we believe a sequel is an opportunity to refine and polish all the concepts developed for the first game. Also, as the team grows from one project to the next, new hires or transfers from within Monolith are expected, and each new person brings a unique perspective to the existing team.

Unified Rendering LOD Part 4

In this series, we've been working on rendering large triangle soup environments. To help accomplish this we divide the environment into chunks, then create reduced-detail versions of the chunks, ensuring that no cracks are introduced in the process of detail reduction.

Last month ("Unified Rendering LOD, Part 3," May 2003), we clipped a triangle soup into two pieces, connecting the pieces with some filler triangles that I called a "seam." We created reduced-detail versions of each half of the input mesh, ensuring that the seam triangles always preserved the manifold between the halves. Now we will extend this clipping method to an arbitrary number of pieces. Then we will be able to render an LOD'd triangle soup using the chunk selection and blending system discussed in Parts 1 and 2 of this series (March and April 2003), back when the system worked only on height fields.

Multiple Clipping Planes

Last month's system only used one clipping plane. You might think that we could just apply that method repeatedly to chop up the input mesh and be done. But some complications arise, so let's look at those.

With only one clipping plane, we create only two chunks of geometry. Computing the seam between these two chunks is relatively straightforward, as we saw last month. But in Part 1 we saw that filling the gaps between each pair of chunks is not enough. In corners where multiple chunks meet, we can have a "pinhole," as seen in Figure 1. With a height field, we might fill these holes by marking the corner vertices of each chunk at preprocess time and at run time by drawing two triangles between these vertices.

But imagine trying to extend this strategy to a 3D rectangular grid of chunks. In 3D, there are two major ways that multiple chunks meet: along cube edges, where four chunks can meet, and at cube corners, where eight chunks can meet. It becomes difficult to see a way that holes can be dynamically filled because there is no longer a coherent concept of "corner vertices" to each block. A cube edge that passes through a mesh can create many "corner vertices," some of which may disappear as the chunk's resolution is adjusted. If a coherent dynamic solution exists, it's messy and probably slow. (The height field seam-drawing code from Part 2 [April 2003] already contained an unsavory amount of confusing code that performs tree traversal to find neighbors, and I'd hate to exacerbate that situation.)

So instead of filling the holes dynamically at run time, we precompute the fill patterns for these holes the same way we precompute the chunk-to-chunk seams. We expand our concept of a seam, allowing seams to touch more than two blocks. We need to precompute and store versions of each seam that cover all possible LOD states of the blocks it touches. Planning out data structures to handle the increased combinatorics for several-chunk seams is a big headache.

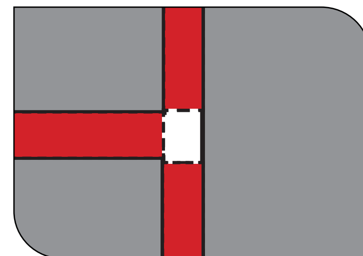
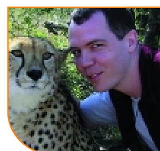


FIGURE 1. Three terrain blocks (gray with black borders) and the seam fills between them (red with dotted borders). Note the hole in the middle. These blocks are drawn with exaggerated gaps; the actual hole would be very small.

Increased Combinatorics

But as it turns out, we need to contend with more than one combinatoric increase. In 3D, we usually can't impose the constraint that two fully diagonal neighbors must always be within one detail level. We'll discuss this next month, but the basic idea is that the constraint reduces your control over the LOD quality level, often by too much. We must give up the one-level-neighbor constraint, which means the combinatorics between neighboring detail levels can grow much larger: we must build seams that tie together chunks that are two or three levels apart.

When I first thought about how to program this system, I envisioned an octree containing all the chunks. To build a seam between some high-resolution blocks and their low-resolution neighbor, I would collapse one level of the octree in the appropriate place, cross-reference the high-resolution seams to build a new seam, and record that result. This process can be applied repeatedly until we exhaust all the combinations, but programming all this is still a headache. (First we need to collapse portions of the octree by one level, choosing them one at a time, then two at a time, then three at a time, and so on; then we need to collapse one portion of the octree by two levels, but the rest of them by only one level, repeating all the previous combi-



JONATHAN BLOW I *Jonathan is a computer games consultant hanging out in Austin. He can be contacted at jon@number-none.com.*

nations; then we need to collapse two portions by two levels, and so on. It just feels nasty, and it would require a lot of the unhappy neighbor-navigation code mentioned previously.)

I was unsatisfied with this solution. I wanted a way to deal with all these combinations that was easy to program and easy to understand, so I could put this LOD manager in the core of my rendering system and have some confidence that it actually works.

Triangle-Centricity

Happily, I came up with a simpler system to accomplish all my goals. Two main observations helped me find the simplifications, both of which came about when I decided to stop thinking about octrees, chunk borders, and seams, and moved to an entirely triangle-centric viewpoint.

First, I realized that any triangle, having only three vertices, can touch at most three chunks simultaneously. Thus, if we ever do anything that cares about the combinatorics of more than three chunks at once, whether at preprocess time or at run time, we're complicating the situation needlessly.

The second observation was that, when remapping the high-resolution seams, we actually don't need much information about which chunks neighbor which others. We only need to know which chunks contribute their geometry to which lower-resolution chunks; then we can use that information to rewrite existing seam triangles, and we get all the combinatorics for free.

A Database of Seams

We can think of each triangle as being a triplet of "vertex specifiers": each specifier tells us which chunk of geometry, and which index inside that chunk, represents the vertex we want. Suppose we have some chunk named A. The vertex specifier for chunk A, index 7 can be written as "A7." A seam triangle connecting chunks A and B might be written as (A7, B10, A8).

Suppose we detail-reduce chunk B into a new chunk C, and vertex 10 of B becomes vertex 3 of C. To help create the corresponding seam, we want to rewrite the above triangle so that it becomes (A7, C3, A8). As long as we perform this step properly for every triangle that contains B in a vertex specifier, we will successfully preserve the manifold. It doesn't matter who the neighbors of A, B, and C happen to be. The fact that seams always tie neighbors together becomes an inductive property, caused by the fact that we only made seams between high-resolution neighbors to begin with. We don't need to worry about maintaining this property between resolutions, because it propagates automatically.

At preprocess time, I maintain a database of all existing seam triangles. First, I split input geometry into chunks and put the resulting high-resolution seams into this database. Then I perform the detail reductions and, for each reduction, execute a rewrite rule on the database. The rewrite rule just searches for all triangles containing a certain chunk in their specifiers, writes new versions of those triangles with the new chunks and indices (such as the B10-to-C3 conversion just mentioned), and adds the new triangle to the database. We repeat this process, always adding new triangles to the database, never removing any.

By the time we've reduced our input geometry to a single low-resolution chunk, the database has computed for us all combinations of all seams between neighbors of all possible resolutions. (To get a feel for this, try it with a simple case with pencil and paper.)

We may not wish to store all these combinations, so we can impose limits. For example, we can tell the database never to create seams between chunks that differ by more than two or three levels of detail. We can even set this limit on a chunk-by-chunk basis, with those decisions arising from an analysis of the input geometry.

I have spoken here of manipulating individual triangles, but to reduce memory and CPU costs in the implementation, I group the triangles into seams as before, with the grouping based on the chunk part of their vertex specifiers. So the "chunk membership" is stored in an array on the seam and used for all triangles within the seam; only the vertex indices are stored per-triangle.

All this makes the preprocessing solution rather tidy. But how do we organize these seams so they can be found quickly at run time? The high-level answer to this is that we just store the seam database wholesale and reload it for run-time use. To draw seams between all the solid-rendered chunks on the screen, we should first make an array of those chunks (which we have already done so that we could render them), and then just tell the database, "Give me all the seams that only touch blocks in this set." Then we render those seams. Simple, easy, done.

Now, "database" is a scary word for engine programmers trying to do fast graphics. One might have nightmarish visions of SQL queries happening inside the render loop. In actuality, because we only need one query at run time, we can set up specialized data structures that help us answer that query quickly; our "database" becomes some arrays or hash tables. But to

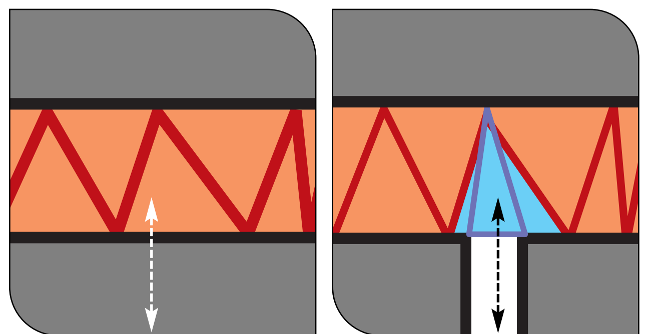


FIGURE 2A (left). Two neighboring chunks, with a seam connecting them (red). We are about to split the lower chunk by clipping it against a plane (green). FIGURE 2B (right). To split the seam, we probably need to subdivide one of the triangles into two (cyan). Once we have two seams, we need to insert a pinhole-filling triangle that connects all three blocks (blue).

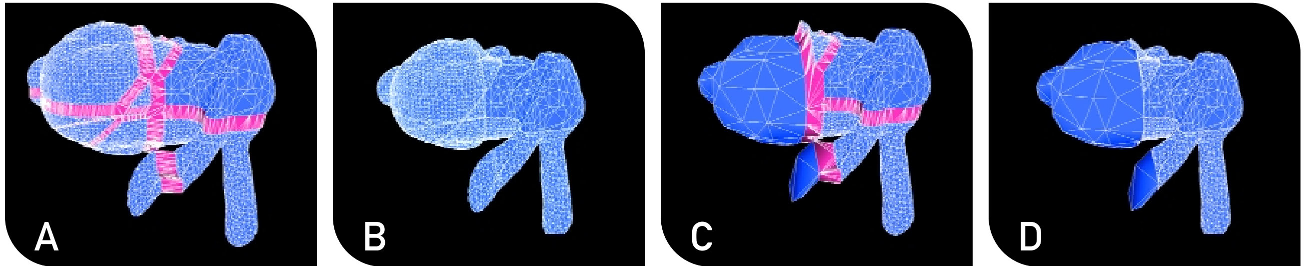


FIGURE 3A. The Stanford Bunny, chopped into arbitrary chunks by four different splitting planes. The original mesh contained 16,000 triangles, but the portion in the upper-right has been reduced by one level of detail. The seam-filling triangles have been drawn in red, and the pieces of the bunny have been pulled apart so that you can see the seams. **FIGURE 3B.** The same geometry as 3a, but rendered more like it would be in an actual game, without the pieces pulled apart. **FIGURE 3C.** Like 3a, but now we have reduced the left half of the mesh by three levels of detail. Note that the seams are still properly filled. **FIGURE 3D.** Like 3c, but without the pieces pulled apart.

maintain simplicity, consider this a problem of "accelerate a database query about vertex specifiers," so try not to fall into the mentality of "store seams in arrays based on neighbors and resolutions," as we did with the height-field renderer.

In this month's sample code, the seams are stored in the database in a single linear array. I performed the database query as follows: First, I mark all the chunks that are being rendered. Then I iterate over every seam in the database and check the chunks it touches (of which, remember, there can be no more than three). If all the chunks are marked, I render the seam, otherwise I don't. After this is done, I unmark all the marked chunks.

This algorithm is $O(m)$, where m is the total number of seams in the database. That's fast enough for the data sizes we are dealing with now, but in a big system it might be a bit slow. By storing seams in arrays on each chunk (which any seam being referenced by multiple arrays), we can reduce the running time to $O(n)$, where n is the actual number of seams you need to draw. Since the task of rendering the seams is itself $O(n)$, it wouldn't help us greatly to try to drive the running time lower. Perhaps I will implement this version of the query next month.

The Moral of the Story

There's a moral to this database story that I would like to pass on. As engine programmers, we're used to thinking about a certain set of concepts and data structures, such as octrees. When approaching a new problem, we tend to apply these concepts first, perhaps disregarding simpler ways of seeing the situation. Even though those data structures have helped us in the past, they may not help us now, and they may serve only to confuse matters. I am reminded of that old proverb "When all you have is a hammer, everything looks like a nail."

Increased Freedom

Now that we use this database rewrite system, neither the preprocess nor the rendering requires an octree. In fact they require very little in the way of data structures. We need only a set of hierarchical bounding volumes for frustum culling and some LOD metric that we can apply to each chunk. That's an amazing amount of freedom, much more than I envisioned when I started this project. That freedom is good; it means anyone using the algorithm will not face many restrictions in how this system must interact with the rest of the renderer.

In fact, nothing in this entire algorithm even cares about the dimensionality of the space we are working in. So if you are some kind of weird physicist running simulations in 11 dimensions and you need a system to perform LOD, maybe this will suit you.

Given all this newfound freedom, I'm going to try something different from what I originally planned. Instead of using a 3D grid of blocks to store the seam, I will employ a system of BSP-style cutting planes, situated at arbitrary orientations. I will then compute these cutting planes based on the density of the scene.

Filling Pinholes

The seam database approach worked so well for LOD generation that I used it for the initial chunk generation as well. I split a chunk into subchunks by applying a single splitting plane and rewriting the seams in the database. Often this will split a seam into two, adding also a single-triangle pinhole-filling seam, as seen in Figure 2. This correctly preserves the manifold for one split, and thus, since we perform one split at a time, it inductively preserves the manifold until we've got all our chunks and are ready to build LODs.

This month's sample code, which you can download from the *Game Developer* web site at www.gdmag.com, contains two different running systems. One of them is the height-field renderer, modified to use the seam database approach. This system serves as a relatively simple introduction to the database, as it doesn't need any of the chunk splitting mentioned above (a height field can be chunked just by applying a window to the array of data).

The second system in the sample code is a new version of the bunny-chopping program, modified now to use an arbitrary number of splitting planes. This program illustrates the BSP-style cutting planes I am talking about, and it serves to verify that the mesh-chunking and pinhole-filling schemes work properly. You can see the results in Figure 3.

Next month we'll look at LOD metrics and discuss methods of choosing splitting planes.

Transition

For most people, change can be difficult or uncomfortable. Unfamiliarity leads to insecurity, and no one likes to feel insecure. New environments and new experiences can be exciting, but those first steps into the unknown are usually accompanied by some degree of apprehension.

The game industry at present is, like the larger economy and job market, somewhat unstable. Job security for the most part is limited, and changing jobs is sometimes inevitable, whether for career advancement or as a result of project termination. Whether your move is local or transcontinental, the transition process is a vital part of successful integration into a new environment, and it is a crucial time for both individual artists and the companies that employ them if the best long-term results are to be achieved.

The following is a collection of advice gathered from a variety of people within the industry, from relative newcomers to company owners, on how to make the transition process as smooth and effective as possible. The first section addresses the experience of the employee, the second looks at the situation from the employer's side.

Illustration by Tris Nerima

The New Employee

In almost every case, the newly arrived employee has more to lose than the company employing them if things don't work out. For the employee, changing jobs every few months isn't the most encouraging thing to have on one's résumé.

For artists making transitions, it's important to distinguish between joining a new company to start a project, or at least arriving when the game is still in its very early stages, and joining a company to work on a game that is nearing completion.

An artist joining a project at or near the beginning is likely to be in a position to contribute to the concept stage, helping shape the game's look, and to also be involved with the setup and organization of the art pipeline (unless the company already has an existing, rigid methodology in place). Involvement at this early stage greatly helps the process of transition, as work becomes more about helping to build initial concept and structure than simply fitting into a preexisting slot. Especially with artists' creative nature, the more restrictions that are put on that creativity, the more difficult and less enjoyable the job can be. Chances are, however, that a change of jobs will not

HAYDEN DUVALL | *Hayden started work in 1987, creating airbrushed artwork for the games industry. Over the next eight years, Hayden continued as a freelance artist and lectured in psychology at Perth College in Scotland. Hayden now lives with his wife, Leah, and their four children, in Garland, Texas, where he works as an artist at 3D Realms. Contact Hayden at duvall_hayden@hotmail.com.*

always bring you in at the start of a project.

Joining a game later on is more difficult, as the artist's contribution is more about content generation and the inevitable reworking of existing assets that characterizes the end of game projects. Stepping into someone else's shoes (who left the vacancy you arrived to fill) or filling a position that has been created at this late stage to make deadlines more achievable puts more pressure on the newcomer who has to adjust to a new work environment as well as having to fit seamlessly with an existing art style.

Joining a project in the role of an art director, art lead, or some other senior role, brings additional pressures due to the challenge of leadership.

Establishing credibility and respect as an artist is central to a rewarding work experience. This is even more crucial if you join a company in a leadership role. There are several ways in which artists and managers can improve their chances of success, compiled from advice from those that have experienced both good and bad transition periods.

Enthusiasm and Attitude

As the new kid, showing enthusiasm about your new company and in particular the game you are working on is vital. This kind of attitude reassures your employers that they made a good choice in hiring you, something that they might be nervous about until you prove yourself. It's also a form of positive feedback for your coworkers, indicating that you like their work and are happy to be part of what they are creating.

As with most things, it's possible to overstep the mark. A new employee is not best advised to charge into a new job with such overwhelming joy that other artists are swamped by your tide of eagerness. When joining a new company, there will be a natural period of acclimation as everyone works out what kind of person you are. When enthusiasm is taken to the extreme, it can be seen as a negative, or interpreted as an attempt to gain favor with the boss, so moderation is advisable.

Respect and Credibility

It's almost impossible for me to say the word "respect" without lapsing into a Marlon Brando parody. And while some game companies may feel like they are operated by the Corleone family, gaining respect for your work as an artist is a legitimate concern.

Respect, however, can be a difficult thing to pin down. In many walks of life, respect is a result of status and a person's achievements. This translates to some extent to the game development industry, and particularly as an artist, a large part of the respect you command comes from the quality of your work. Every artist you work with has his or her particular skills and areas of expertise. It's part of the natural process of interpersonal assessment that your work will be used to inform other artists' opinions of you, and vice versa.

As a professional you will have to earn the respect of those that you work with through your conduct and attitude. Respect is most effectively built on a mutual basis, and demonstrating that you respect the work of others around you is an important step in this direction.

Acceptance and Change

Being accepted into the team is about more than just the allocation of work and a name on the credits. As with any social grouping, there are unwritten rules about how the group works, who fits into which role, how different personalities interact, and the inevitable degree of company politics to negotiate.

One important way to gain acceptance is to ensure that your priority on arrival is to learn rather than attempting to instigate changes. Whether you're a junior artist or the newly arrived creative director learning about the team, the game, and how everyone currently operates is certainly the place to start. The fact that you can see improvements that can be made from day one does not necessarily mean that immediately pointing those needs out is the most productive approach to take, even if you are in a position to do so.

It's true that as a group of professionals, everyone should be able to accept suggested changes that will benefit the project, even if they come from the new guy. But human nature cannot be discounted, so a "settling in" period is advisable. This time can also be used to build a more complete picture of your new company and new position. Having a solid feel for these areas will help you suggest changes in the most effective manner.

Conflict and Criticism

While it's best to avoid workplace beatings whenever possible, artists have the difficult job of dealing in an area that is essentially subjective in nature. For programmers, code tends to either work correctly or not. Artists, however, are subject to opinions of a more ethereal nature. Whether you are giving opinions or receiving them, you need to be tactful, measured, and able to accept opinions with which you don't agree.

New artists especially can be subject to a large amount of feedback (some of it negative, while they find their footing). Newcomers must accept this as guidance more than criticism as they learn new stylistic and methodological processes. There is also a tendency for senior members of a team to feel that they need to be seen as having input, which often comes in the form of requested changes. This is hard to counter but will usually diminish noticeably once you have been there for a while. Overall, it's vital to take criticism well regardless of its origin, and to discuss changes rather than argue about them.

First Impressions

Don't arrive at your new job with the intention of putting your stamp of ownership on the game. Every developer I have talked to has expressed dislike of the prima donna attitude sometimes displayed by those who believe they are

special. Even if you are special, making a game always has to be a team effort. Anyone who works against this, especially if they are new, is unlikely to be appreciated.

Dangerous comparisons. One of the most counterproductive things a new employee can do is to constantly harp back to a previous job, especially when the comparisons are unfavorable to one's present employer. Statements like "At my last place that was all taken care of in the editor, which was so much better than what you have here" don't help, and referring to your present employer as "you" rather than "we" is never a good thing.

Mouthing off. Our industry may be large in terms of revenue, but it's surprisingly small and incestuous in terms of the workforce. With this in mind, it's never a good idea to bad-mouth people you may have worked with previously, chances are that you'll be working with someone who knows them and may well not share your opinion. This is not to say that you need to be the epitome of sweetness and light, it's simply a case of using discretion when telling your story about "John and the transsexual hooker."

Politics and power struggles. If your new company has a lot of that kind of thing going on, there is no easy way to avoid office politics. If you are an art lead, it's important not to distance yourself from the art team by taking the "executive lounge" route. The best leaders have always led from the front and by example, which is particularly applicable to the collaborative nature of game art. If you are the new artist on the team, it's important that you are not projecting an image of being in competition with the other artists; once this begins to happen, the team loses its coherence in the face of one-upmanship, which always builds resentment.

The Employer's Role

Similar to the points made about the new employee's role, there are several things that an employer needs to consider when dealing with a new employee's transitional period. The following advice has been collected mainly from the experiences of those I have spoken with as they themselves have moved into new jobs over the years. The following illustrates the reciprocal importance of a new employee's transitional period. The new employers also need to help make the transition smooth if they want to get the best out of their new hire.

Be prepared. Preparation is a basic point, but it's often overlooked when developers are deeply occupied in actually making a game. Little is more frustrating for a new employee, ready and raring to go in a new position, than turning up for work and finding that a computer hasn't even been ordered, let alone set up. My informal research suggests over half of the artists beginning work have faced delays ranging from a few days to over a month while waiting for hardware.

Moreover, a top-of-the-line PC is of little use if the none of the software the new artist needs is on it. Add to this the availability of scanners, digital cameras, graphics tablets, and basic furniture needs, and it becomes clear that employers need to be on the case before the artist steps through the doors, ready to work. Not only do delays like this waste the artist's and the company's time, but first impressions matter for employers too.

Beyond software and hardware needs, an employer needs to have informed the relevant people (ideally the whole company) of any new arrival so that there is a plan in place ready to streamline the integration process. Sitting in front of a screen, waiting for someone to figure out what to do with you is not the best introduction to your new job, but unfortunately, it happens all too often. Because there are no hard lines between what's right and what's wrong in art, it's even more important that a new artist be given the necessary information and guidance to learn what he needs to. Details such as acceptable polygon count will be dependent on the specifics of the game, its engine, and the platforms it's running on, and should be communicated as soon as possible.

Be open and inclusive. Sometimes game companies (especially larger ones) treat a new employee as if they are some kind of intruder. Secrecy is usually a tool of managerial manipulation and an indication of a lack of trust. While that's another topic entirely, employers need to be as open and honest as possible if they expect their new employee to trust them and be honest in return. I am not talking about disclosure of the owner's salary, but things that need to be made clear are important issues, such as an explanation of the real chain of command. Every company, even those that deny it, have some form of hierarchy; the more covertly this system operates, the harder it is for a new employee to understand who reports to whom and more importantly, who actually has the final word. Pretending that person A is the one to sign off on your new character design is a pointless exercise if in reality, his or her superior can decide it's unacceptable later that week.

Employers must also remember to include a new employee as much as possible. Not knowing the routine can isolate new staff, so the employer should actively attempt to involve them in as much as possible both during and after work if necessary (for example, if they are new to the area).

Transition Is Short, Success Is Long

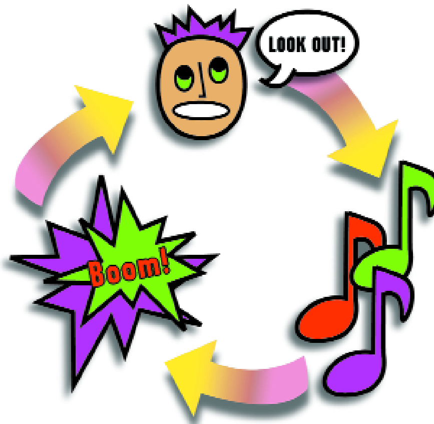
While the sensitive artist stereotype is largely fictitious, and no one that I know would burst into tears if they found it difficult to transition into a new job (well, maybe I can think of one person who might), both employer and employee can learn from the experiences of others to make a transition period more successful. Hopefully the considerations presented here will make your next transition, whether from the employer's or the employee's perspective, more profitable in the long run by helping avoid common mistakes and frustrations that often get things off to a bad start.

Managing the Nonlinear Mix

Game audio elements are typically created and mastered independently of one another. Within a game, any number of these independent sound effects, voice tracks, and music tracks are played simultaneously. In a situation where the order and layering of these elements is not predetermined but rather determined by player interaction, a nonlinear mix occurs. In this situation, problems may arise causing balance issues between sound effects, music, and dialogue.

Furthermore, unforeseen combinations of sounds that are dominant in similar areas of the frequency spectrum can create a mix that sounds "muddy" or "cluttered." While the actions of the game player determine the timing of certain audio events — and in many cases the frequency at which those events occur — the sound designer has control over each element's amplitude, its spectral content, and the texture or timbre of the overall game soundtrack. With some forethought given to texture, spectral, and level management, sound designers can reduce the common problems that result from a nonlinear mix.

Texture management. Texture management is the organization of the audio assets. By examining the overall style and audio requirements for the game and subsequently its individual levels or subsections, it's possible to determine the game's overall sonic texture. Some questions to consider include: Should the ambient sound effects dominate, or is the gameplay music-driven? What role will spoken dialogue play, and what is the priority of that role? Will triggered, event-driven sounds interrupt the dialogue or disrupt the music tracks? The answers to these types of questions will help determine the overall texture of the soundtrack. Based on these answers you can focus the audio design for each segment of the game to highlight those key audio elements or textures deemed most important.



Level management. Level management refers to the amplitude of individual sonic elements or classes of sonic elements within the overall soundtrack of the game. Fundamentally, level management describes the volume levels of the sound effects, music, and dialogue, as well as the overall dynamic range of the game's soundtrack (that is, the difference between the loudest and softest sounds in the game). More in-depth level management includes looking at the amplitudes of sound element groups. For example, how loud are the footsteps in relation to the ambient sound effects in relation to an NPCs' sound effects in relation to the NPCs' dialogue? So while texture management determines what sounds should be occurring on a per-level basis, and the player determines when those sounds are playing, level management establishes relative amplitudes for those sounds.

The implementation of level management occurs at several stages within the game development process. The first set of amplitude decisions are made when the sound elements are mastered as individual files. Level management occurs again when the sounds are integrated into the game during the audio coding process. Playback levels are set to be executed by the audio engine during gameplay. Finally, some game titles delegate a portion of audio level management to the game player, giving the player overall balance control between voice, music, and sound effects from a sound options menu.

Spectral management. Spectral management relates to the allocation or consideration of the spectral content of sounds or groups of sonic elements within the game. Some nonlinear mix problems can be reduced or eliminated through the use of creative equalization or pre-allocated frequency bandwidths on specific audio elements. For example, in areas with constant music, heavy ambient sound effects, or both, an equalization curve could be applied to those elements allowing more room in the spectrum for additional event-based sound effects or dialogue. Attenuation in the lower-mid-range frequencies could allow for additional intelligibility in the dialogue track. Another notch in the upper end of the spectrum could help accommodate transient sound effects that might be triggered.

While the concepts of texture, level, and spectral management are not new to the audio mix environment, careful attention must be paid to these concepts in the early stages of sound design. Early and thorough planning will reduce later headaches and result in a more cohesive and well balanced nonlinear mix.

MIKE VERRETTE | *Mike is the audio director for Wicked Noise and member of the Game Audio Network Guild. When not managing his spectrum he can be reached at mike@wickednoise.com.*

The Hobgoblin of Little Minds

A minor trivia game for your enjoyment. What is "the hobgoblin of little minds"? Two points for the correct answer. Score another three points if you know who said it. And if you know the entire quotation, give yourself a big five points.

The answer?

"A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines." — Ralph Waldo Emerson

I only scored the first two points myself. When I looked up the full quotation, I was surprised to see that Emerson was insightful enough to condemn not consistency, but a foolish consistency.

That's trumping information. Be consistent, but don't be foolishly consistent.

Is that one of the 400 Rules of game design? I think it's too vague. Foolish is as foolish does. So I'm turning to you for help in clarifying it. I've been admonished for taking the obvious direction with my choice of the rules I've published here, and I think rightfully so. I started with the easier ones that were hard to argue with, thinking that was a good way to prime the pump with rules. But now that there's a steady flow, it's time to risk a little more controversy.

It's obvious that some consistency is a good thing. You wouldn't want a game that showed every enemy unit in red and every friendly unit in green — except for a friendly medic unit with a big red cross that you blew up the first time you saw it approaching. And consistency for control interfaces is important too, you wouldn't want one part of the game to have one interface and then suddenly change to a different one.

Or would you?

If you have a platform game and you're happily running and jumping but then get the magic hat that lets you fly, don't you need to change the interface, at least a little? If you switch from a first-person view to a 3D strategic map view, you have to change the way the player selects a target — don't you?

NEVERWINTER NIGHTS may not always be consistent, but it does have lots of hobgoblins!

So perhaps the rule is "Be consistent in interface as long as the context of what the player controls remains constant."

And yet if you can remain consistent even when the context of control has changed — perhaps by turning that jump button into a flap button — isn't that even better?

Let's look at it from a slightly different perspective. One reader suggested to me that a good rule would be "Be consistent in player feedback." If smashing a crate (to take an original example) gives you something useful, don't have the 200th crate you find blow up in your face and take away a life.

That seems obvious. But at the recent Game Developers Conference, I had an interesting conversation with Mark Cerny, a designer who has been a design contributor to games totaling over $1 billion in sales (that's billion with a *b*). Regarding that very issue of rewards, he told me that he thinks that every once in a great while you should blow the player up for no reason. My first reaction was, "He's lost his marbles!" But having reason to believe Mark is quite aware of where his marbles are, I started to think about the player's experience.

Let's say the player is an hour or two into a game and has opened 200 crates, all with good stuff in them. The initial thrill has probably long since worn off, and opening crates has become rote, even boring. Then, with no warning, the next crate blows up in the player's face, costing a life. What's the player going to think? The player won't stop opening crates — after all, 200 had rewards in them. But the player will approach every subsequent crate with more care and more excitement. When crate number 600 blows up, the player may even think there's a pattern. But Mark insisted there must be no pattern. I like that — it fits in with the AI rules I mentioned in my January 2003 column ("AI Without Pain") about a little randomness and the power of suggestion. And it suggests that the really interesting and useful guidance to a game designer about consistency is when to break that consistency.

Or to put it another way, when does too much consistency become foolish?

I have some ideas about that, but I'd like to turn to you, the readers. Send me some one-sentence imperative rules that a fellow designer can actually implement, and perhaps some associated simple exceptions. Please refrain from long discourses, I'm hoping to boil these down into a few simple, clear rules and information about when to trump them, and why. I'll discuss the most provocative ones in a future column. Who knows, we might even find one consistent rule about consistency. That would be a good thing — wouldn't it?

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

melissa federoff

Improving Games with User Testing: Getting Better Data Earlier

Illustration by Dominic Bugatto

Most game designers do not acquire major feedback on their products until beta, when quality-assurance representation on a given product increases. QA testers are trying to break the game: they're finding bugs, they're finding balancing issues, they may even be finding major gameplay problems. But even if they can offer feedback on design problems, the game is meant to ship in a matter of a few short months, so it might be too late to fix most of them.

How can game teams get design feedback earlier, when the game isn't in final form yet? Several possible methods come from a discipline called usability. Central to the concept of usability is the evaluation of users, or target audience. By borrowing concepts from the field of usability and applying them to games, designers can get ongoing data on how to improve their games right from the consumer.

What Is Usability?

Usability is a field of study where a product is tested on actual users for efficiency, effectiveness, and satisfaction throughout development. These three measures help to determine if the design goals of a product are being met. In the case of productivity software, testing these elements of usability helps to ensure that the user can meet the product's goals, that the user can achieve them in a way that expends the least amount of energy, and that the software provides an overall satisfactory experience to the user. Often in productivity software, efficiency and effectiveness are the focus of testing, because these are the main selling points for businesses, the major buyers of the products.

How Usability Applies to Games

With games, we have an entirely different situation from productivity software. By selling to individuals rather than businesses, there is an incredible amount of competition, both from the vast amount of games on the market and from all the other types of entertainment consumers can choose to buy and engage in at any given time. We are also selling an experience, not a tool. We want players to be able to interact with the game's interface efficiently and effectively, and we want them to play through levels as close to our ideal path as possible. Above all else, we want them to have fun.

Using usability terms, then, we can and should be testing for effectiveness and efficiency, but for games we want to focus our efforts on measuring satisfaction. These usability measures can help us to judge whether players are able to use the controller adequately, check their status in the game correctly, use menus without frustration, make their way through levels successfully, learn skills as they progress, be challenged appropriately, and have a desire to keep playing because they are experiencing the type of entertainment and amount of fun they expect.

Test Early, Test Often

Game designers are no different from other types of designers. They generally think they can judge the quality of their own project and anticipate how their audience will perceive it. Though sometimes it is possible to guess how someone else will interact with a product, design choices can only be verified through testing.

A full-scale production is too large of an investment to base on hunches or artistic desires. Commercial game development is not creating art for art's sake; it's creating a product to sell to a consumer. Therefore, developers need to test, at every stage of development, whether the product is reaching its goals.

Volunteer participant Lily Childs plays RTX Red Rock during an in-house test at LucasArts.

Usability vs. Quality Assurance

In order to assess whether a design is working well, it needs to be tested on the target audience — those who are likely to buy and use it after it is released. QA testers are not representative of the target audience of a game, because they are professional game players. The vast majority of game consumers do not get the level of exposure to games that a QA tester has. Therefore, testers are unlikely to interact with a game in the way a nonprofessional player would. Also, since QA testers are often a part of the actual game team, they, like designers, are often too close to a game to judge it in an objective way.

MELISSA FEDEROFF | *Melissa works for LucasArts Entertainment Company. She began her study of the usability of games as a graduate student in the MIME (Master's in Immersive Mediated Environments) program at Indiana University. Her thesis, which is an expanded version of this article, can be found at www.melissafederoff.com/thesis.html.*

Getting accurate impressions of a design is critical, and so is getting the feedback in a timely way. Acquiring information about a design late in development can mean one of two things: either the necessary changes are not made and the product is not improved, or changes are made and the product is derailed from its production cycle. If feedback is acquired about design decisions as they are being made, before large investments of time are made in them, the product will more easily be improved.

Implementing Usability Techniques

Usability offers the game development industry different methods from which to choose for evaluating games. These methods vary in the type and quality of data they produce based on the resources available to implement them.

Some methods — such as expert evaluations — do not require the cost of labs and participant compensation, but they do require the cost of having a usability professional run the tests. Other methods, like play-testing, can be done in-house to avoid participant compensation and coordination costs, but still require employee time.

No matter what method is chosen, usability testing costs money up-front, but investing in it can save time by avoiding costly mistakes and should increase the overall quality of products if performed correctly.

Microsoft Game Studios User-Testing Group Usability Lab: observer side.

Microsoft Game Studios User-Testing Group Usability Lab: participant side.

Beginning Steps

Before engaging in any usability tests, it's best to incorporate extra time into the production cycle for testing and fixes so that changes can be made as problems are found. Doing multiple rounds of testing is important for finding new problems as they arise, as well as for verifying that any changes that have been made are working.

Concept testing. Ideas can be tested even before they hit the digital realm. Reading story scripts or showing concept art to members of the identified target audience can help display what ideas are resonating and what ideas are not getting across the way the designer intended.

Prototyping. After an idea is concept tested, it can be prototyped. This can be on paper, such as a story flowchart or an interface mock-up, or it can actually involve creating a mini playable version of the game. A playable version could involve designing a level that incorporates all of the important components of

the game. When a playable prototype is developed, any technological or strategic concepts can be tested for viability. But the benefits go far beyond that.

With a prototype, all critical game elements can be analyzed before they are invested in too greatly. These elements are all created during preproduction with a smaller team, which should save the cost of developing them. Any major obstacles should be found early so that fixes or alternate paths can be determined before the production is fully underway. Since the preproduction team is small, it also gives the core team time to gel together and figure out their processes before taking on the added responsibilities of team management.

Play-testing. Once a playable build exists for a game, play-testing can begin. During a play-test, someone plays a game and offers feedback on their experience with it. This can be done with in-house participants or out-of-house participants and can range from formal to informal.

For instance, at LucasArts there is an informal in-house play-test group. Anytime a team wants feedback on their product, they can call on a group of volunteers to play their game for them. Recently, play-testers were requested to play an original action-adventure title, RTX RED ROCK. Volunteers were scheduled for sessions at a level designer's desk and asked to play a specific level of the game for two hours while members of the game team watched and took notes. The participants were asked to think out loud while they played so that the team could gain better insight into what choices they were making as they progressed through the assigned level.

The game team felt that the feedback improved the levels greatly, and plan to do more testing in the future. Overall, the cost and time involved was low, and the impact on the game was high. Mainly, the team laments not having begun these tests earlier and not having enough time to address everything the tests revealed about the game.

Play-testing can help teams in seeing how actual players interact with the product. The team is too close to the design to see objectively, and the data collected from these exercises can be very enlightening.

Lily Childs thinks aloud while playing RTX RED ROCK during an in-house LucasArts test, as team members Harley Baldwin, Tim Miller, and Shara Miller take notes.

Offering the opportunity for open-ended feedback at the end of a play-test can elicit even more insight into the experience with the product. Asking questions such as "What did you like best (or least) about the game?" can allow teams to anticipate what is or is not fun about the game at a point when gameplay can still be adjusted.

Advanced Measures

In-house play-testing provides valuable feedback, but the information gathered still isn't as accurate or helpful as it could be if greater resources were available. This type of testing is not truly reflective of a target audience, because even if the participants fall into the correct demographic (genre of game, age, gender, and so on), they are still coming to the test with much more game knowledge than an average consumer would have. They know more about games and game development in general, and they are within the company, which means they inevitably are bringing some preconceived notions about the product with them to the play-testing session.

Play-testing with outside users. Using outside participants instead of people in-house requires greater resources. First of all, compensation is required to encourage people to attend sessions. Money, copies of games, or other company memorabilia can serve this purpose. The second expense is the overall effort it takes to schedule participants in a timely way.

Before testing begins, the facility in which the participant is to be tested has to be determined. They can simply play at a designer's desk, although such a setup potentially skews the data. Participants may feel compelled to try to please the designer, since he or she will be present, which can alter how they evaluate the product and what sentiments they choose to express while thinking aloud. The designer may also find it extremely difficult not to intervene in the test in some way. To keep designers from altering the test process and results, they ideally

Microsoft Game Studios User-Testing Group Play-test Lab cubicle.

should observe from a separate room either through one-way glass or on video.

Play-testing doesn't have to be a one-on-one experience with a designer. A participant can play a portion of the game for a given time frame and then offer feedback through a questionnaire. Once participants get a taste of the game, they can offer a subjective perception about the product's overall fun factor.

Questionnaires. Creating a scientifically sound questionnaire is a tricky business and ideally requires a psychology expert. Depending on how a question is phrased, it will generate a different answer. In order to get the most accurate data possible, questions have to be developed with the least likelihood of leading the person to answer in an anticipated or predictable way. Getting someone to create questionnaires and analyze the data is expensive. The data resulting from this type of test, though, will be quantitative and better able to guide the design revision process. Knowing the percentage of people who felt positively or negatively toward certain facets of the game can help the team prioritize which changes to make in the amount of time available.

Structured usability evaluations. While play-testers can work en masse in a big enough lab, structured usability tests are run individually. In a structured usability test, information can be acquired about efficiency and effectiveness, rather than satisfaction as in play-tests. If designers want to know whether users are learning required tasks as they are playing, whether tasks and or levels can be completed without too much confusion, or whether a control scheme feels intuitive, structured usability testing is a great method to use.

To use this method, tasks to be performed are established prior to testing. Usually, someone familiar with the game will run through the tasks and establish an ideal time for each one. Then, a user is asked to perform them while thinking aloud. As the participants progress, the following is noted: whether they can successfully accomplish each task, how long they take to accomplish it, and anything they say or do to indicate how they are working to achieve each goal. In the usability field, six to eight users can uncover the vast majority of usability problems with each task. While it's promising that so few users are needed to find the majority of design problems, there is no guarantee that the solution implemented for any problem is successful unless it too is tested.

For game development, a variation called RITE (Rapid Iterative Testing and Evaluation) might solve some of the time issues with finding and resolving usability issues using the standard method. In this technique, developed by Microsoft Game Studios User-Testing Group (see For More Information), a participant engages in particular tasks and then the design is changed immediately based on the results before another user is run. This way, a set of users are not experiencing the same problems over and over again, but rather verifying whether past issues have been resolved. This method yields fast results and makes the most of each participant's involvement, but requires the development team to have time as testing occurs to make the ongoing changes.

Expert evaluations. Beyond user-testing

FOR MORE INFORMATION

GAME-SPECIFIC USABILITY RESOURCES

Microsoft Game Studios User-Testing Group:
www.microsoft.com/play-test/publication

Federoff, Melissa. "Heuristics and Usability Guidelines for the Creation and Evaluation of Fun in Video Games." Master's thesis, Indiana University, 2002.
www.melissafederoff.com/thesis.html

GENERAL USABILITY RESOURCES

Dumas, J., and J. C. Redish. A Practical Guide to Usability Testing. Norwood, N.J.: Ablex, 1993.

Nielsen, J., and R. Mack. Usability Inspection Methods. New York: John Wiley and Sons, 1994.

Norman, Donald. The Design of Everyday Things. New York: Doubleday, 1990.

Jakob Nielsen's Web Site
http://useit.com

ONLINE GUIDE TO USABILITY RESOURCES

www.usabilityfirst.com

methods there are expert evaluation methods, which can be utilized by usability professionals. These evaluations do not require users, but do require someone who knows how to anticipate user behavior. While these methods do not yield data with the same amount of validity as user testing, they can help to uncover usability problems in a faster, cheaper way when necessary.

One of the most common expert evaluation methods used by usability professionals is the heuristic evaluation. Heuristics are agreed-upon standards that are used to evaluate a design. To do a heuristic evaluation, the usability professional makes one pass through the product to become familiar with it, and then makes a second pass to determine whether it is meeting or failing each heuristic. Jakob Nielsen has created heuristics for software interfaces, but an agreed-upon list of heuristics does not exist yet for games. Table 1 shows one possible list of heuristics for games.

Another expert usability evaluation method that could be applied to games is the cognitive walkthrough. During this technique the usability professional walks through a scenario and tells a convincing story about whether the determined path for players would be the one they would actually take. This would be a particularly strong method to use with games that have a linear structure to make sure that the expected path for the players is the one they will likely follow.

Raising the Bar

The field of usability offers developers many methods to determine the efficiency, effectiveness, and satisfaction of product designs throughout development. The data resulting from the empirical testing of users can help designers to make informed decisions and improve the overall quality of their games. Many games are already taking advantage of usability testing to improve player experience and satisfaction, which will ultimately raise the bar for all games in terms of consumers' expectations.

GAME HEURISTICS

#	Category	Heuristic
1	Game Interface	Controls should be customizable and default to industry-standard settings
2	Game Interface	Controls should be intuitive and mapped in a natural way
3	Game Interface	Minimize control options
4	Game Interface	The interface should be as non-intrusive as possible
5	Game Interface	For PC games, consider hiding the main computer interface during gameplay
6	Game Interface	A player should always be able to identify their score/status in the game
7	Game Interface	Follow the trends set by the gaming community to shorten the learning curve
8	Game Interface	Interfaces should be consistent in control, color, typography, and dialog design
9	Game Interface	Minimize the menu layers of an interface
10	Game Interface	Use sound to provide meaningful feedback
11	Game Interface	Do not expect the user to read a manual
12	Game Interface	Provide means for error prevention and recovery through the use of warning messages
13	Game Interface	Players should be able to save games in different states
14	Game Interface	Art should speak to its function
15	Game Mechanics	Mechanics should feel natural and have correct weight and momentum
16	Game Mechanics	Feedback should be given immediately to display user control
17	Game Mechanics	Get the player involved quickly and easily
18	Gameplay	A clear, overriding goal of the game should be presented early
19	Gameplay	There should be variable difficulty and multiple goals for each level
20	Gameplay	"A good game should be easy to learn and hard to master" (Nolan Bushnell)
21	Gameplay	The game should have an unexpected outcome
22	Gameplay	Artificial intelligence should be reasonable yet unpredictable
23	Gameplay	Gameplay should be balanced so that there is no definite way to win
24	Gameplay	Play should be fair
25	Gameplay	The game should give hints, but not too many
26	Gameplay	The game should give rewards
27	Gameplay	Pace the game to apply pressure to, but not frustrate the player
28	Gameplay	Provide an interesting and absorbing tutorial
29	Gameplay	Allow players to build content
30	Gameplay	Make the game replayable
31	Gameplay	Create a great storyline
32	Gameplay	There must not be any single optimal winning strategy
33	Gameplay	Use visual and audio effects to arouse interest
34	Gameplay	Include a lot of interactive props for the player to interact with
35	Gameplay	Teach skills early that you expect the players to use later
36	Gameplay	Design for multiple paths through the game
37	Gameplay	Players should be rewarded with the acquisition of skill
38	Gameplay	Build as though the world is going on whether your character is there or not
39	Gameplay	If the game cannot be mode-less, it should feel mode-less to the player

TABLE 1. Game heuristics developed during a thesis case study performed by the author. They were compiled after reviewing relevant literature and observing and interviewing a development team. They are a starting point for discussion; further research is required in order to verify them. The project is available online in its entirety at www.melissafederoff.com/thesis.html.

Beautiful, Yet Friendly Part 2: The Hardware Pipe

Last month, we reviewed performance at a high level, and we looked at how level design and environmental interactions affect it. Since most of the theory was also explained in the first part, I strongly suggest that readers get familiar with the concepts introduced last month before reading this section: you'll need to know when and what to optimize before making any use of 'how' to optimize.

We saw last month that meshes could be transform-bound or fill-bound. I've given a more complete picture of the possibility space here through the generic hardware pipe shown in figure 3D_Pipe_2C.

If you are data-bound, then the amount of data transferred is probably also causing transform problems (too many vertices) and/or fill problems (too much texture data). Data-related problems generally arise through a collection of objects, and not by single objects in isolation. If you find that you're clogging the BUS, then you should redistribute your texture and vertex densities across your scene (See last month's description of this). If you are CPU-bound, then it's basically out of your hands: the programming team will need to take a hard look at their code.

Optimizing transform-bound meshes

If design wants marching armies of zombies attacking the player, you'll need to make sure they don't put the renderer (and artist) on death row by minimizing their transform cost.

We saw previously that the cost of a transform-bound mesh is:

Transform cost ≈ Vertex Count * Transform Complexity

Hence, we need to reduce the transform complexity or the number of vertices. You can somewhat reduce the transform complexity by plucking out bones you don't really need, but you should consider using a less expensive type of transform first. If you can approximate a morph target accurately enough with a few bones, you'll save on transform complexity. If you engine is optimized for non-weighted vertex blending – where vertices can be affected by only one bone – see if you can't substitute your vertex-weighted mesh with a clever distribution of bones that take no vertex weights. In any case, take the time to consult with the programmers as they may have insights on better transform techniques you can use to lower your transform complexity.

Space-farers make babies

Before you go on plucking vertices out of your mesh, I'll let you in on a secret: the vertex counts in your typical modeling package don't reflect reality. As they travel down the pipeline, vertices get split, and re-split, ad nauseaum. Vertex splits adversely affect transform-bound meshes by adding spatially redundant vertices to transform. In theory, vertices can get split as many times as they touch triangles, but in practice, total vertex counts generally double or triple. If you're smart about it, you can lower this

GUILLAUME PROVOST | *Originally hired at the age of 17 as a lowly system programmer writing BIOS kernels for banks, Guillaume has been trying to redeem himself ever since. He now works as a 3D graphics programmer at Pseudo Interactive and sits on his local Toronto IGDA advisory board. You can contact him at depth@keops.com.*

split ratio dramatically and make your mesh a whole lot more performance-friendly without removing a single vertex.

Let's first examine the nature of the splits. As I mentioned before, graphic hardware thinks in terms of surfaces1, not objects. So the first vertices that get 'split' are those lying on the boundaries of two different surfaces. Think of it in your head as: a vertex cannot be shared across multiple materials (figure ALL_SPLITS_1C – slot B)

Similarly, renderers typically do not allow vertices to share polygons with different smoothing groups, or vertices that have different UV Coordinates for different triangles. So vertices that lie on the boundaries of two different smoothing groups are split, and vertices that have multiple UV coordinates (they lie on the boundaries of discontinuities in UV Space) will also cause splits2 (figure ALL_SPLITS_1C – slots C and D).

There are several ways to simply minimize individual types of splits. Intelligently combining and stitching textures together, for example, can help minimize material based-splits.

UV space discontinuities tend to be a bit trickier. Mapping an element without any UV break means that you'll have to find either an axis of symmetry, or at the very least a 'wrapping point' on your mesh.

If you can get away with using mapping generators, such as planar, cylindrical or cubic mappings, you can minimize or altogether eliminate UV space discontinuities. Ball-jointed hips and shoulders, for example, can make the resulting arm and leg elements ideal candidates for such techniques.

If you need to split the mesh in UV space, both 3DStudio Max 5.0™ and Maya™ have elaborate UV mapping tools that permit you to stitch UV seams in order to minimize the damage3. Its generally well worth spending the time to optimize your mapping in UV space since it will also both simplify your texturing pass and minimize the texture space you will actually need for the object. When no axis of symmetry existed, we found that treating the texture as pieces of cloth that you 'sow' up worked well to minimize UV splits when texturing humanoids. (Figure UV_SPLIT_2C)

If you are building a performance critical mesh, then it's probably best that you fine-tune and optimize the smoothing groups by hand. Remember that the goal isn't to minimize the number of different smoothing groups, but rather the number of boundaries that separate those smoothing groups. You can also 'fake' smoothing groups by using discrete color changes in the texture applied to it, avoiding splits altogether, although this may not result in the visual quality you are attempting to achieve.

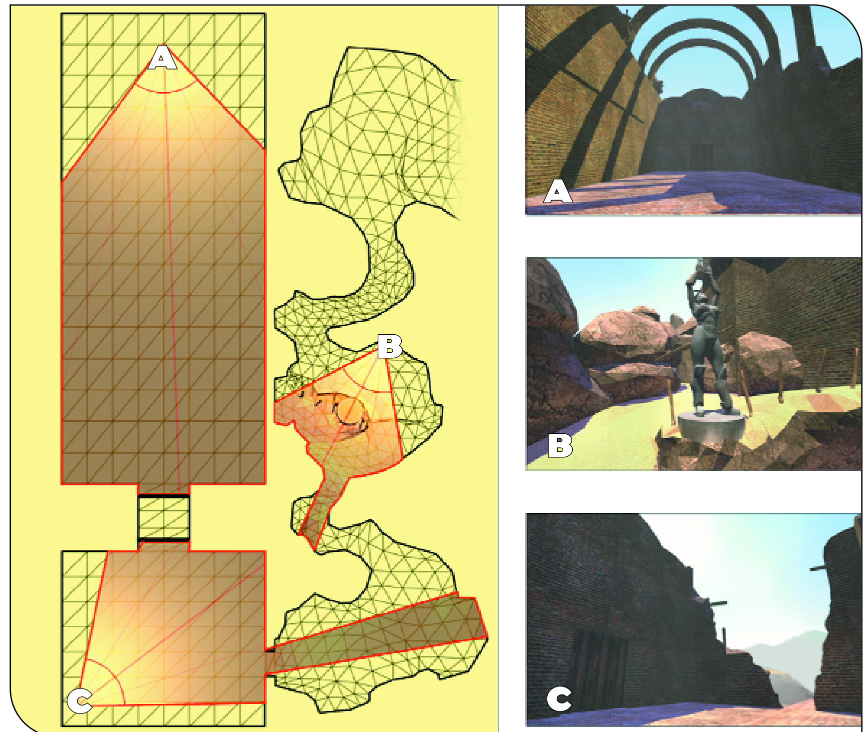


FIGURE 1. The visibility spectrum in a portal-enabled visibility engine. In camera A and B, the visibility spectrum is closed off using a door that disables the portal. In camera C, transition zones keep the player from seeing into the next area. The very small visibility spectrum in C lets us place a highly detailed statue that we could not afford in the other areas.

Another way to look at it altogether is to 'reuse' vertex splits. For example I said earlier that renderers allow one material per vertex and one smoothing group per vertex. In other words, if you have a smoothing group and a material id group that occupy the same set of faces, they'll get split only once. The same goes for UV discontinuities: if they occur at smoothing group boundaries, then they won't cause an extra split to occur.

For the record, if your mesh is definitely transform-bound then it is generally more important for you to save on vertex splits than to save on texture memory. If that means authoring an extra texture for the mesh in order to get rid of individual diffuse-color based

materials or UV breaks, then it's a fair trade-off.

This brings us to normal maps and the general – and increasingly popular – concept of using high detail meshes to render out game content. Normal maps are textures for which every texel represents a normal instead of a color. Since they give extremely fine control over the shading of a mesh, you can replicate smoothing groups and add a whole lot of extra shading detail by using them. Since normal maps are generally mapped using the same UV coordinate set as the existing diffuse texture, they do not cause extra vertex splits to occur, and are in effect cheaper for transform-bound meshes – and much better looking – than smoothing groups.

Unfortunately, normal maps cannot really be drawn by hand: they require specialized tools to generate them and higher resolution detail meshes if you want to take full advantage of their potential. Because of the involved pixel operations required to support them, they are also not supported on all hardware platforms.

At the end of the day, make sure you absolutely avoid checkerboard-like material switches, where you consistently cycle between materials and, unless your programmers specifically support it, setting whole objects as flat-shaded by having every individual faces be a different smoothing group (Figure FLAT-SHADE_1C).

If they strip well, GPUs love them

When I originally set out writing this article, I was actually naïve enough to think I could safely cover solid guidelines that covered all mainstream console systems, and all recent PC-based video-cards without encountering critical system specific guidelines. Then reality hit: I was, as always, overly optimistic.

Some systems don't support indexed primitives, and some don't have a T&L transform cache. In either case, your surfaces' transform cost will be significantly affected by their 'strip-friendliness'. If your hardware does support both, then strip-friendliness is less of a performance issue.

A triangle strip is a triangular representation some systems use in order to avoid transforming a vertex multiple times if it's shared amongst one or more triangle(s). In a triangle strip, the first three vertices form a triangle, but every successive vertex also forms a triangle with its two predecessors. When graphic processors draw these strips, they only need to transform an additional vertex per triangle, effectively sharing the transform cost of the vertices with the last (and next) triangle.

Stripping algorithms 'close a strip' (effectively increasing transform time) when there are no vertices they can choose in order to form a new triangle. This typically happens at tension points (Figure STRIP_1C), where a single vertex is shared amongst a very high number of triangles (8 or more triangles).4

Since tension points are always connected to a series of very thin triangles, avoiding sliver triangles and distributing your vertex density as equally as possible on the surface of your mesh will generally help the stripping process.

Most good triangle stripping algorithms will automatically re-triangulate triangles lying on the same plane5, but they cannot reorient edges binding faces on different planes.

Transform bound meshes conquered

Okay, so knowing about all these technical details can make a transform-bound mesh up to three times more efficient if you're smart about what you're doing, but it's a lot of work. Remember to always ask yourself if you need to optimize a mesh before you go on with all the hard work. Otherwise, use these techniques opportunistically. In the end, having a tool that helps visualize where vertex splits occur is tantamount to building truly optimized meshes. As a summary of things to look out for, here's an optimization checklist for transform-bound meshes:

Build one or more LOD (Level of Detail) meshes for the object

Use as few bones and vertices as you can, try decreasing the transform complexity

Use as few material surfaces as you can get away with; consider texturing your mesh instead of using several different diffuse colors.

Use UV generators to minimize UV discontinuities

Get rid of smoothing group breaks you don't really need, or use discreet color changes to fake them, or use a normal map

Match the remaining material boundaries, UV-space boundaries and smoothing group boundaries

Validate your invisible edges and look out for tension points.

Avoid sliver triangles and try to make the vertex density as uniform as possible across the surface of the mesh.

If you think that your mesh is fill-bound instead of transform-bound, then do not do any of the above. Combining materials into a single texture applied to a fill-bound mesh, for example, might actually hurt your performance by causing cache misses to occur more frequently.

Optimizing Fill-Bound meshes

We saw earlier that the cost associated with drawing fill-bound meshes was a function of three things:

Fill cost ≈ pixel size * draw complexity * texel density

You can't exactly make your walls any smaller than they are, but you

should avoid overlaying several large surfaces within the same visibility space. A typical example of this would be to have an entire room's wall covered with an aquarium (the back-wall and the glass window create two layers), or successive sky-wide layers of geometry to simulate a cloudy day. Transparent and additive geometry tends to accumulate on-screen, potentially creating several large layers of geometry the renderer needs to draw, and creating a fill-related bottleneck.

If your export pipeline supports double-sided materials, be wary of using them arbitrarily on large surfaces: you can easily double your fill rendering costs if you are forcing the render to draw wall segments that should be culled. On some platforms, back-face culling is not an integral part of the drawing process, and culling individual polygons becomes a very expensive task; if you are authoring content for such platforms, you should ensure that walls that don't need back faces don't have any.

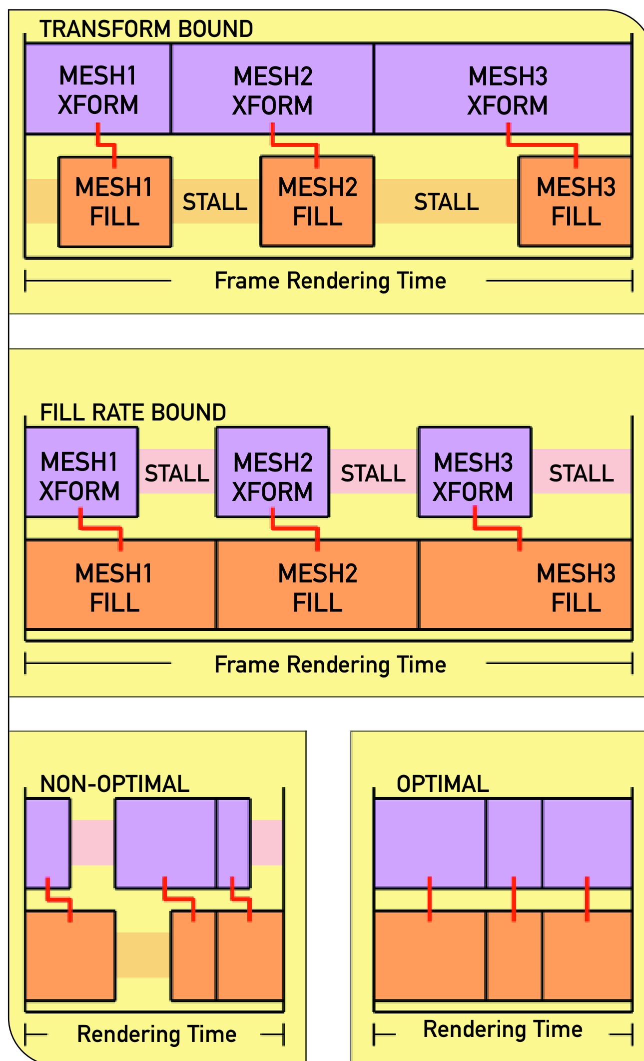


FIGURE 2. Transform/fill parallelism.

The bigger the triangles, the less texture space you want to address. Unfortunately, in practice, meshes that take up the largest portion of screen space also tend to also gobble up the most texture space, and so they are prime targets for being fill-related bottlenecks.

There are two things you should do to minimize your texture space: make sure you are using/generating mip-maps, and choose your texture formats and size intelligently.

A simple table illustrates simple savings you can do by making smart choices about your texture formats (Figure DXTC_2C). Note that, if your textures are smaller than 32x32 texels, it's probably not a good idea to palletize them, since the cost associated with uploading and setting up the palette is larger than just using the un-palletized version. If your hardware supports native compression formats, such as DXT1 (DirectX Texture Compression), it's probably a good idea to use them over palettes.

If you can get away with using diffuse colors only on a fill-bound surface, so much the better: on several platforms, drawing un-textured surfaces is faster then drawing textured ones.

As a general rule we mentioned earlier that it was a fair trade-off to sacrifice texture space in order to prevent UV splits in transform-bound meshes. When you mesh is fill-bound, the contrary rule applies: if splitting the vertices in UV space will help you save texture space, it's also a fair tradeoff.

Finally, making conservative decisions on the nature of the materials you apply to fill-bound meshes will pay off in performance. The number of texture passes and the complexity of their material properties will always be the biggest factor at play when dealing with fill-bound surfaces.

Texels miss the boat

Some of us deal with the 'crème de la crème' when it comes to hardware, but the vast majority of us need to contend with market realities. In the console market, teams get to really push a system to its limits, but they are also stuck with those limits for a very significant time period.

If you count yourself in that situation, then chances are you need to take texel cache coherency into account.

Graphic Processors typically draw triangles by filling the linear, horizontal pixel strips that shape them up in screen-space. Almost all current hardware can do this by 'stamping' several pixels at a time, greatly decreasing the time it takes to fill the triangle.

For every textured pixel the card draws, it needs to retrieve a certain amount of texels from its associated texture6. It does this through a texel cache, which is just basically a scratchpad on which the card can pastes texture blocks. Every-time the card draws a new set of pixels it looks into its cache. If the texels it needs are already present in the 'scratchpad', then everything proceeds without a hitch. If some texels it needs are not in the cache, then the card needs to read in new texture chunk(s) and place them in the cache before it can proceed with drawing. This is a texture cache miss.

A good texel cache coherency means few texture cache misses occur when drawing a surface. A bad texel cache coherency will significantly increase the time it takes to draw a surface. Most PC-based systems and a few of the current high-end consoles will automatically ensure a good texel cache coherency by choosing the proper mip level at every pixel they draw.7 But other systems rely on the fact that the texel density across the surface area of a mesh in geometric space is constant for their pixel choice to be correct.

On such systems, non-uniform texel densities will cause the card to 'jump' in texture space from pixel to pixel. This can cause both severe texture aliasing problems, and will consistently cause texture cache misses to occur as the card tries to fetch texels that are not in its 'scratchpad'.

As an artist, you can solve both those visual artifacts and performance problems by ensuring you uniformly distribute texel density across your mesh (Figure TEXDENS_2C). You can do this by ensuring the size and shape of your faces in UV space is roughly proportional to their counterparts in geometrical space. This is a concept that makes sense on an artistic perspective as well: if a face is bigger, it should get more texture detail (a larger UV space coverage) than a smaller one.

The concept extends to objects too: if an object is smaller, it's likely to be smaller on screen as well, and should get a smaller (less-detailed) texture.

Fill-Bound surfaces Conquered

Here follows the list of things to look out for when constructing fill-bound geometry:

Build mip-maps for all textures

Shy away from large surfaces with complex material properties (such as bump-maps, and glossy materials)

Do not overlay several very large transparent or additive layers.

Don't make large wall/ceiling segment double-sided unless you absolutely have to. If your engine doesn't support back-face culling, make sure to get rid of large unnecessary back faces.

Choose your texture formats intelligently to save texture space. If you do not have access to compression formats such as DXT1, see if you can't palletize textures.

Use small texture swatches or diffuse materials instead of larger textures, even at the expense of vertex splits.

Tweak your UV maps to distribute your texel density as uniformly as possible across the surface.

The good news about fill-bound surfaces is that, although adding more vertices probably won't help8, it probably won't make much of an impact until your vertex density is high enough for your mesh to become transform-bound.

Conclusion

If your head is spinning by now, follow Douglas Adams motto: don't panic. Although there is a lot more to performance-friendly content than meets the eye, building efficient content can become an intuitive, natural process.

Whether they are vertices, texels, objects or textures, it's more about uniformly distributing them then about plucking out detail. This is a very powerfully intuitive concept: things that are smaller on screen should get less detail then things that are bigger on screen.

Programmers can always optimize their code to go just a little bit faster. But there's a hardware limit they can never cross without sacrificing visual quality. If you are pushing the limits of your system, chances are that it is your content – not code – that drives the frame-rate in your game.

1 A surface, as reviewed last month, is the set of all faces in an object that share the same material properties.

2 If you have objects with multiple UV channels, the splits will occur successively through every channel.

3 Maya™ even has a UV-space vertex counter, which should reflect the number of vertices in your mesh after UV splits.

4 Certain renderers support triangle fans. Fans make tension points very efficient. But given that current hardware only supports one type of primitive per surface, they tend to rarely be supported in practice.

5 Verifying this with the programmers is safer.

6 Since the pixels are unlikely to fall directly on a texel, renderers typically set up video-hardware for bi-linear filtering, which fetches and blends four texels for each texture involved.

7 Some systems also support Tri-linear filtering, which blends texels across mip-map levels at every pixel for maximum image quality. Tri-linear filtering (also referred as linear mip-filtering) pulls in more texels, and is significantly more expensive for fill-bound surfaces.

8 This isn't entirely always true. Very large polygons can trash the texture cache on some systems, effectively increasing fill-time. In such cases, tesselating the polygons will actually help.

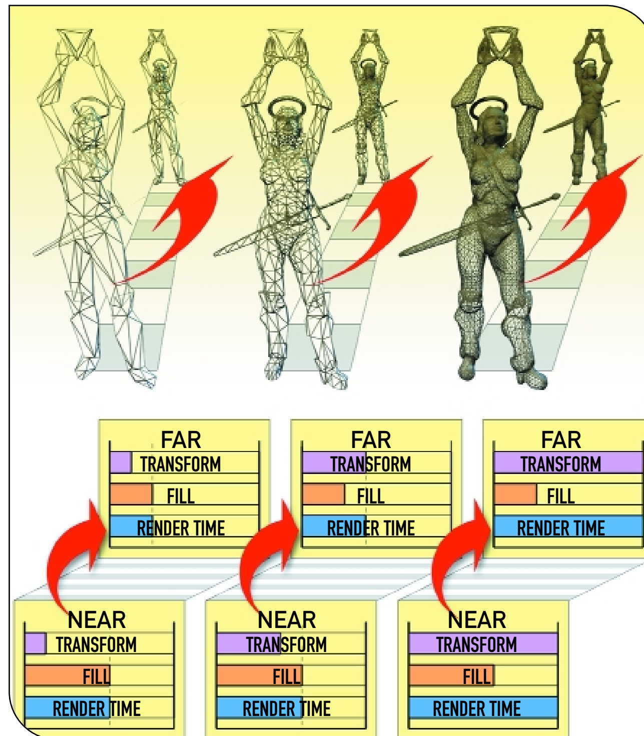


FIGURE 3. The fill cost of a mesh decreases with distance, while the transform cost does not. This can cause meshes to shift from being fill-bound to being transform-bound as they recede into the distance (center). If a mesh has a very high vertex density, then its cost may stay the same regardless of distance (right). To reduce the associated transform cost of a mesh in conjunction with its diminishing fill costs, lower-level-of-detail meshes are used (left).

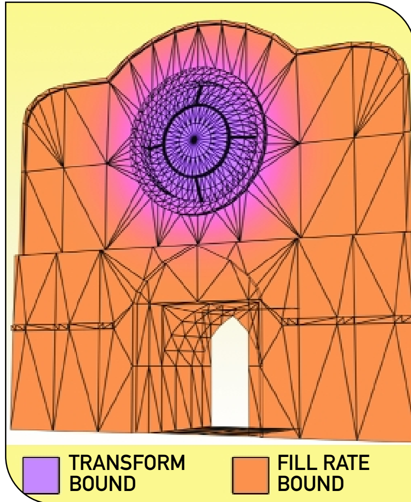


FIGURE 4. Non-uniform density mesh.

ACKNOWLEDGEMENTS

Thanks to Danny Oros for providing both the illustrations and several rounds of feedback. Thanks also to Benoit Miller from Matrox, Sébastien Dominé and Sim Dietrich from Nvidia, Guennadi Riguer from ATI, and Ryan McLean from MagiTech, for kindly answering questions and reviewing the article. And last but not least, thanks to the rest of the Pseudo team, for playing the guinea pigs on this article.

Insomniac Games' Ratchet & Clank

GAME DATA

PUBLISHER: Sony Computer Entertainment America
NUMBER OF FULL-TIME DEVELOPERS: 40
NUMBER OF CONTRACTORS: 1
LENGTH OF DEVELOPMENT: 18 months
RELEASE DATE: November 5, 2002
TARGET PLATFORM: Playstation 2
DEVELOPMENT HARDWARE: PS2 Dev Tools, PCs: avg. dual 800MHz–1.2GHz with 1GB RAM
DEVELOPMENT SOFTWARE USED: Insomniac's own tools suite, Maya, Photoshop, ProDG, MS Visual Studio, CodeWright, Deep Paint, ProTools, Sound Forge, Premiere, Illustrator
NOTABLE TECHNOLOGIES: Much of the engine technology was developed in-house, but some very important renderers were developed by Naughty Dog; sound licensed from Sony

The scene: Twenty developers lounging on a sun-drenched porch overlooking Barham Boulevard in Los Angeles, drinks in hand, enjoying the warm breeze and listening to traffic rumble by below. The occasion: Our first post-SPYRO brainstorming meeting.

It was late spring 2000, and even though we were still in production for SPYRO: YEAR OF THE DRAGON (our last SPYRO), we knew we had to start planning for our first PS2 project. Our problem was twofold: we had decided not to develop any more SPYRO games, and we were deciding whether we wanted to stay with the platform-action genre. It's a familiar scenario for game developers: the road is wide open, but figuring out which direction to travel is excruciating.

We had meeting after meeting trying to narrow down the choices — and with 20 people involved, things got tense and sometimes depressing. I was driving hard to move us away from the platform genre because Al Hastings, our vice president of technology, had very astutely suggested that this was the perfect opportunity not only to expand our abilities but to address other niches in the console market currently overlooked by U.S. developers.

After coming up with and discarding countless ideas, we settled on a concept best described as a dark adventure. We wanted to try a game with a bit more realism and immersion than our previous efforts. This meant moving away from bright environments, cartoony characters, and platform mechanics. This also meant creating a macro design and story that were far deeper than those of the SPYRO series.

We called the concept "I5" (for Insomniac game #5), and the main character was a human girl with a staff. She would fight with the staff as well as use it to activate magic with special katas — martial arts moves performed using directional input. There was a strong Mayan influence to the overall look of the game, and the characters and environments we planned were more realistic than anything we had attempted since our first game, 1996's DISRUPTOR.

We pitched our game idea to SCEA and were fortunate to strike a deal very early in preproduction. Once we had Sony's backing, our preproduction team dove in and began working on PS2 technology, final macro design, and all of the elements that would help us create our first playable.

Within a couple of months, however, it was clear that things weren't going well.

First, we couldn't nail down the main character. She was too cartoony, and then too mundane; the colors we chose ended up looking weird on-screen, and we couldn't get the proportions right. In the past, proportion had never been a problem, since we had always worked

TED PRICE | *Ted is president and founder of Insomniac Games.*

with nonhuman characters. But we quickly realized that it's easier to spot flaws in human characters than in nonhuman ones. Even though our main character eventually looked acceptable, she still lacked that *je ne sais quoi* which would make her stand out.

Then there was the hardware. We were making the jump from PSX to PS2 in very little time, and Al Hastings was shouldering the entire burden with some help from Mark Cerny, who had written the original VU code used on the first-ever PS2 engine. Al and T.J. Bordelon, tools programmer, were, at the time, trying desperately to get the engine and tools to the point where the artists could use them to build and prototype environments and characters. Looking back, I can't believe they actually got everything to work, and work well, in a matter of months. Still, the technology was not yet state-of-the-art, and we all wondered how it would fare against the second generation of PS2 titles.

But the worst part of the process was the entire team's ambivalence about the project. No one was truly excited about the game or where it was heading. We were making it work through sheer effort. My job was to be the concept's champion, but maintaining a positive demeanor was proving more and more difficult. Morale was at its lowest in Insomniac's nine-year history.

We eventually ground out a first playable, and while it wasn't bad, it

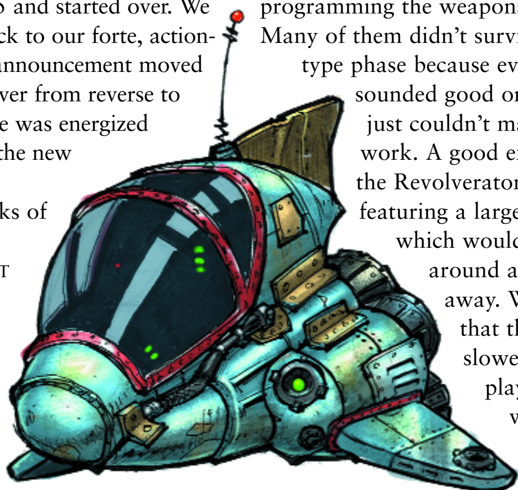
Some early concept sketches for Ratchet.

wasn't great either. And we wanted something great. Our Sony producers, who were very polite about their reservations, confirmed our feelings. Nonetheless, they had reservations. At one point Connie Booth, our SCEA executive producer, suggested that we might want to rethink the direction we were taking. While being very clear that Sony would support us with whatever we decided, she pointed out that not only would the PS2 adventure category be crowded upon our planned release date, she also believed that we were no longer playing to our team's strengths.

After digesting her words, Al Hastings, Brian Hastings — Insomniac's vice president of programming — and I (the three partners in the company) did some soul searching and realized that Connie was right. By pushing on, we could release a solid adventure game, one that might even do well. But slogging through another year of developing a game no one was excited about would kill the team.

So on March 20, 2001, we stopped preproduction of I5 and started over. We would be going back to our forte, action-platforming. This announcement moved the team's mood lever from reverse to overdrive. Everyone was energized and excited about the new prospects.

Within two weeks of this decision, we developed RATCHET & CLANK's basic concept. In a matter of days, Dave Guertin, our lead character designer, nailed the two main characters, and soon we were brainstorming on the weapons and gadgets that players would be using.



Once we got started, we never looked back. That isn't to say problems didn't exist during the process, but it was the best and most enjoyable production experience we've had at Insomniac.

What Went Right

1. Prototyping. We had been prototyping gameplay since SPYRO THE DRAGON, but never to the extent that we did with RATCHET & CLANK. The game featured more than 35 weapons and gadgets, all of which had to be fun to use. The big problem we faced was that every weapon and gadget was woven into the macro design and the story. If we had to pull one out during production, the macro design would collapse, which would be disastrous for the production schedule.

We spent three months building and programming the weapons and gadgets. Many of them didn't survive the prototype phase because even though they sounded good on paper, we just couldn't make them work. A good example was the Revolverator, a weapon featuring a large drill bit which would spin enemies around and fling them away. We discovered that the spinning slowed down gameplay, and that it was difficult to hit enemies, since the collision for the drill bit had to be narrow to be believable. Another good idea on paper was the Mackerel 1000, a fish that would be a replacement for Ratchet's wrench. It sounded funny, but when we put it in the game the humor lasted for about three seconds.

We also prototyped enemy layouts and behavior to a much greater extent on this project. The majority of our enemies were well tested and tuned before each level went into production. This process saved us a massive amount of time, since we only built final models and did final coding once we were sure that the enemies would work. Conversely, on the Spyro series we were always ripping things out and starting over during production, since we rarely prototyped gameplay. With RATCHET & CLANK, and for all of our future projects, gameplay prototyping has now become an ongoing process.

Finally, to clearly establish the look of the game, we used our I5 engine to prototype two of the game's planned environments before we had the real RATCHET & CLANK technology up and running. It was all smoke and mirrors, but it allowed us to show on-screen what we imagined the final game would look like and put to rest a lot of our own fears about whether or not the game would stand out visually.

2. Sharing technology with Naughty Dog. Shortly after we decided to start over, Jason Rubin, Naughty Dog's co-founder, called me and asked if we'd be interested in checking out the technology they developed for JAK & DAXTER. He explained that Naughty Dog didn't want anything from us other than a gentlemen's agreement to

share with them any improvements we made to whatever we borrowed plus any of our own technology we felt like sharing. In an industry as competitive as ours, things like this just don't happen.

We went over to Naughty Dog's offices and took a look, particularly at their background renderer. They had developed some incredible proprietary techniques to render smoothly transitioning levels of detail and instanced objects very quickly. We brought the code back to our offices, spent some time getting a handle on their techniques, and then we were up and running with a much more powerful environment engine.

Needless to say, Naughty Dog's generosity gave us a huge leg up and allowed us to draw the enormous vistas in the game. In return, we've shared with them any technology in which they were interested, but so far we've been the clear beneficiary of the arrangement.

3. Setting reasonable design goals. Even though the concept behind RATCHET & CLANK was ambitious for us (integrating RPG elements into an action-platformer), we were careful not to cram too much stuff into the initial design.

We had never made a game before where we didn't have to axe one or more levels at some point in the production process because we were out of time. The RATCHET & CLANK macro design was more complex, so we couldn't afford to rip out a level at the last moment. Sony had created a tremendous marketing campaign that relied on a specific release date, so missing our delivery dates was not an option. Plus, we were already releasing pretty late in the year, and to miss one week of precious pre-Christmas sales would prove very costly.

For these reasons, we planned the game layout much more carefully than we had on past titles. We had a pretty good idea of how long it would take to build each level, but we also knew that plenty would go wrong during the production process. So even though we had time to do 20 levels, we cut back to 18 at the very beginning.

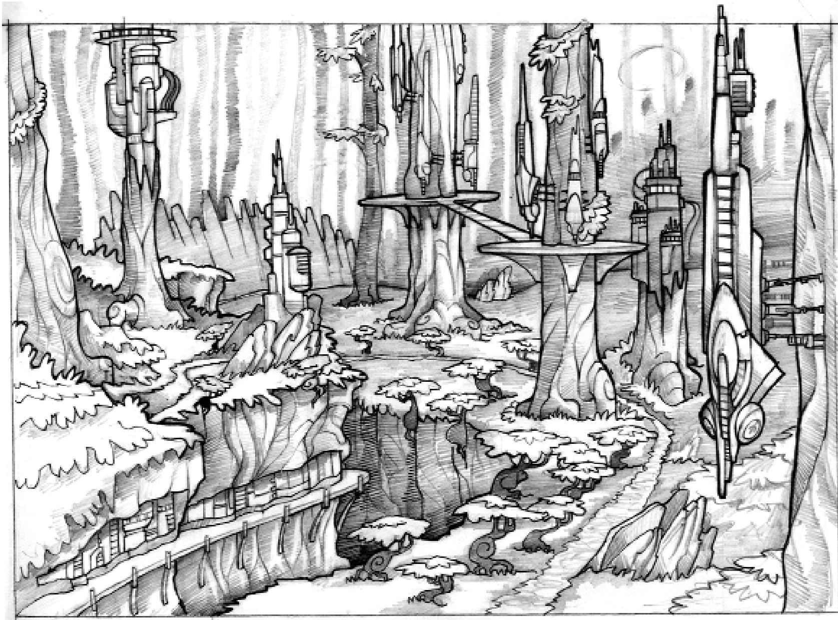
We also made sure that nothing went into the design unless we were very sure that it was going to work. Early prototyping was key here, but so was an attitude of general restraint. There were a few wild concepts that everyone was excited about, but had we integrated them into the macro, the project probably would have slipped. Ultimately we were able to put about 90 percent of what we planned into the game — a record for us.

4. Focus testing. Most games go through focus testing at some point. Publishers and developers alike want to see how people react to the game and whether it's too difficult or too easy. Because it's the best way to tune the gameplay, we've focus-tested our games since the first SPYRO. But with RATCHET & CLANK we went overboard.

We had four major focus tests during production. Each focus test featured another 25 percent of the game until we were testing the full game at alpha. More than 200 consumers got to play the game before release, and the feedback we collected was invaluable. By recording and charting data from the game, we were able to tune item prices, adjust challenge difficulty, and change monetary rewards. Without this exhaustive process the game would probably have been unplayable.

Just as important, though, was the fact that each focus test forced us to get the game working. Along with the other deadlines it sometimes felt that we were

One of the game's early production design sketches.



A concept sketch for a location that would eventually serve as planet Eudora.

always in crunch mode. The gameplay programmers in particular lived a nightmare existence between fixing bugs for the next focus disc and trying to move ahead with the new levels. But the constant burns kept us on track and on schedule. Given RATCHET & CLANK's scope and complexity, if we had waited until the end of the project to burn playable discs, the bug list would have been overwhelming and we would have missed our ship date by months.

5. Collaborative design. Everyone in the company has always been free to contribute creatively to the projects. It's not a requirement, but for those who are interested it's an opportunity to affect the direction our games take. Programmers are encouraged to contribute to story, artists are asked for ideas on design, and so on. During RATCHET & CLANK, a large percentage of the team contributed ideas outside of their particular areas of expertise, making the game one of the deepest and most varied titles we've developed.

This does not imply that we design by consensus. There's a solid structure in place to ensure that we adhere to the macro design and remain consistent with the game's "flavor." But adopting an approach that encourages design participation gives us a real wealth of creativity from which to draw while enhancing the sense of ownership everyone feels in our games.

What Went Wrong

1. Poor disc-burning process. Making the switch from CD-ROMs on the PSX to DVDs on the PS2 sounded like it would be easy. After all, we survived the challenge of recording PSX discs with quirky burners and non-intuitive burning software. What didn't account for was the incredible amount of time that building and burning the DVDs would take.

We had to first transfer the code and data to the PC on which we would generate the files necessary to create a playable disc. Next we'd have to transfer the files to the burner PC. Then the burner software would have to create a disc image, and finally we could burn the disc. By the end of the project we were working with 4GB of data. Combining those steps with slow connections and a burner that we had to use at only double speed to prevent errors, the entire process took more than four hours to generate one disc. And there were many, many places along the way where something could go wrong, forcing us to start over again.

There were countless instances where a level would be out of memory or someone would change the memory card format, breaking everything. But we wouldn't know about it until the final disc had popped out of the tray and had been booted up on a test station. Two mistakes like this would cost an entire day.

So why didn't we change the process? Based on our PSX-burning experience, where the system was extremely finicky, when we had things working on the PS2 we didn't want to touch it and risk breaking everything. This was especially true near the end of the project.

As a result, a few of us didn't go home for days at a time near the end of the project. I remember promising our testers that if our first gold burns worked, I would do circuits of the office singing Britney Spears songs as loud as I could. Fortunately for everyone in the office, they didn't.

The result of our disc-burning pain is that we've now completely overhauled our system. We believe we've halved the overall disc production time for our current project.

A panoramic shot of Kyzil Plateau, on Planet Veldin.

Rendering of the Gagdgetron Headquarters, located on Kalebo III.

2. Late start on cinematics. RATCHET & CLANK has a much more lengthy and involved story than any of our previous projects. Oliver Wade, our animation director, compiled the scenes and found that we've got more than 60 minutes of movies. Even though most of them are about 30 seconds long, that's a lot of animation time. The problem was that we only gave our team of seven animators five months to animate them. That doesn't sound too bad until you consider that the animators creating the movies were also responsible for the in-game animations. Therefore they effectively had 2 and a half months. If you don't include weekends, that's about 10 seconds per animator per day. And that's a lot.

Fortunately, the animators had finished most of the in-game animations by the time the movies were in full swing. But it was still a real challenge. Furthermore, animating the scenes was just the first step. We had to add programmatic and 2D effects and convert many of the animations into MPEGs before alpha, which stretched many people to the limit.

We got such a late start because we had to finalize the story, write the scripts, audition the actors, record the dialogue, and put the final sound files together before starting the animation. It helped somewhat that we took an iterative approach — starting animations as soon as the first scenes were recorded — but in general the tardy start created a lot of stress.

3. Immense level designs. Even though we tempered our ambitions for the macro design, sometimes we cut loose and created some absolutely huge level designs. We had a habit of wanting to make each level better than the last, and a few times this tendency resulted in layouts that made the artists want to kill the designers.

Early on, we didn't have a good understanding of what "too big" meant. The first level designs we created were reasonable, but then we decided that we really needed to show off the power of the RATCHET technology. We also had some ambitious gameplay ideas involving a fight on a moving train and a hoverboard race. This resulted in the Metropolis and Blackwater City levels, two of the biggest in the game. When the artists saw the layouts they said, "Are you nuts? There's no way we can build this in six weeks!" So the designers went back to the designs and tried to edit them, but the levels still ended up being massive.

To the artists' and gameplay programmers' credit they made these and other huge levels work, and they did it on time. And to the designers' credit, they continued to find better and better ways to put more gameplay into smaller areas without sacrificing creativity. In the end, our level design ambitions pushed the limits of time and resources we had allotted.

Out of this stress came a more team-oriented approach to level design, where we now involve a large number of people — artists, programmers, sound engineers, and others — earlier in the design process. Whether or not levels in our future games will be smaller remains to be seen. But with more people involved at the beginning stages, we can find solutions sooner to balancing the need for gameplay space in levels with the time we have available to build them.

4. Maya issues. Maya is a superb tool for building polygonal environments and characters, and it's also great for animation and for prototyping particle effects, rendering, and many other things. However, early in the project we had decided to use Maya as our construction, texturing, lighting, and gameplay placement tool. We had abandoned our in-house tool, Karma, which we had used previously to do gameplay placement, texturing, and lighting. What we didn't realize was that with the size of our levels, we would push Maya past the breaking point.

Even though we set people up with dual 1.2GHz Dells with superfast graphics cards and a gigabyte of RAM, Maya would still chug and frequently crash

whenever our levels got up to around 40MB. And forget making all 500K polygons in a level visible. Fortunately, Al Hastings and T.J. Bdelon worked valiantly to create a suite of plug-ins and tools that worked with the Maya API. This solution didn't always prevent the crashes that plagued the artists or the occasionally corrupted level, but it kept us running and allowed us to create finished levels every six weeks.

While Maya has always been and probably will still be our first choice for art creation, we're moving back to our original approach of using proprietary tools for things like gameplay setup, lighting, and texturing.

5. Localization woes. From the beginning we planned to include the NTSC and PAL versions of the game on one disc. This plan created two problems for us. First, we had to send all of our assets to Europe for localization in French, Italian, German, and Spanish as early as possible. In most cases this meant pre-alpha, which really put the squeeze on the animators who were working on the movies. Second, we knew that we would end up fixing both functionality and localization bugs at the same time. We anticipated that this would create even more chaos during the last few weeks before we went gold. And we were right.

Surprisingly, the biggest nightmare for us was the text localization. We had made the decision to allow subtitles for all of the movie scenes; plus we had a lot of text for the help system and a ton for the menus. We used spreadsheet databases to ensure some organization for all of the text (as opposed to entering localized text in the actual code, which we did on the SPYRO series), and this allowed us make updates and changes quickly. But the system was also prone to user error when cutting and pasting changes into the database.

Because we were still fixing TRC (technical requirement checklist) bugs — things like memory card messages — we were making text changes up to a couple of weeks before gold. We had also added some text late in the process to support some of our postgame features.

We made mistakes, and the localization folks in Europe made mistakes when putting fixes into the database. In addition, it took forever to transfer our discs to Europe once they were burned (eight hours to FTP if nothing crashed, 24 hours for a courier). These facts combined meant that we were still desperately trying to resolve some TRC issues hours before the gold disc was due. Fortunately, the game shipped on-time in all territories, but I think it prematurely aged our producer in Europe, as well as a few of us here.

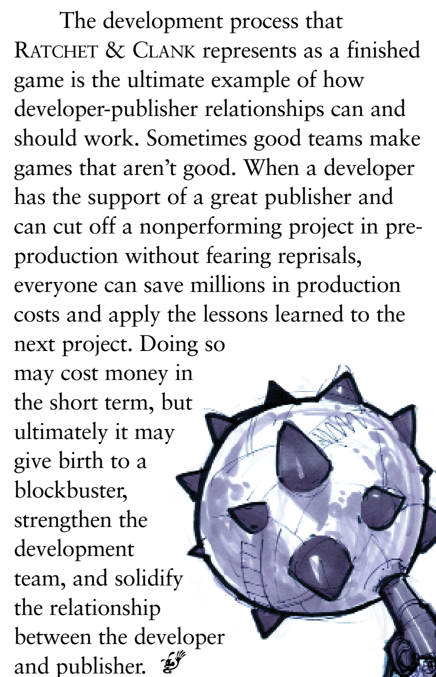
The Will to Kill

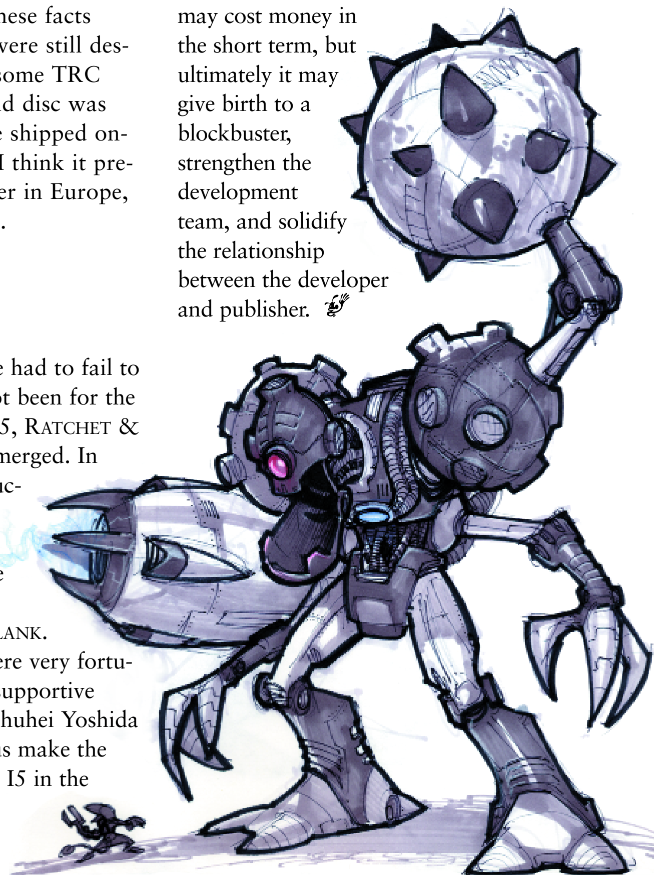
With this project, we had to fail to succeed. Had it not been for the pain we went through on I5, RATCHET & CLANK might have never emerged. In the six months of preproduction on I5 we learned how to make games on the PS2, and we were able to hit the ground running when we switched to RATCHET & CLANK.

Most importantly, we were very fortunate to have an extremely supportive publisher in Sony. SCEA's Shuhei Yoshida and Connie Booth helped us make the agonizing decision to shoot I5 in the head. But they made sure we understood that if we wanted to continue down that dark path of developing I5 for release, they would still support us.

Furthermore, Sony never once threatened to cancel the I5 project or sever our relationship. Instead, they helped us to develop what Mark Cerny calls "the will to kill" — meaning we grew the balls to voluntarily throw out everything we had worked so hard on for six months and start over.

The development process that RATCHET & CLANK represents as a finished game is the ultimate example of how developer-publisher relationships can and should work. Sometimes good teams make games that aren't good. When a developer has the support of a great publisher and can cut off a nonperforming project in preproduction without fearing reprisals, everyone can save millions in production costs and apply the lessons learned to the next project. Doing so may cost money in the short term, but ultimately it may give birth to a blockbuster, strengthen the development team, and solidify the relationship between the developer and publisher.





Gaming Our Way to Our Better Future

New Organizations and Pioneering Initiatives Are Creating New Outlets for Game Developers

Illustration by Tyrone McCarthy

You never know when serendipity will strike. One day you're trying to figure out what to do, the next moment the phone rings and you're on a plane to New York to meet with a major foundation. That was my situation two years ago, and now I'm in the middle of what could be a potentially explosive new outlet for game developers and publishers.

The result of that phone call and airline trip was VIRTUAL U, a college administration simulation developed by Trevor Chan and Enlight Software for the Alfred P. Sloan Foundation. Today that single game, in use at dozens of colleges, has pushed me onto the Serious Games Initiative at the Woodrow Wilson Center. Serious Games is establishing a series of projects leading up to the sustained creation of policy and management games for government and nongovernmental organizations. Dave Rejeski, leader of the Serious Games Initiative, calls it "gaming our way to a better future," a better future for the world in general. In VIRTUAL U's case, the better future is someone's college education. In a game developer's case, it's about a better future for the individual, the industry, and our society.

The idea behind both projects is that games, or in these cases, game-based simulations built by game developers, can be a compelling new generation of entertaining and effective experiential tools for people dealing with complex systems and systems management issues. The bigger scope is that games as a media form are serving to disseminate information, knowledge, and critical thinking. This epiphany isn't new to game fans or developers, but it's quietly becoming such to the world at large, its governments, and other major forces.

I am not proposing that until VIRTUAL U came along a game developer hadn't earned substantial revenues developing a game for non-entertainment purposes, nor would I suggest game skills haven't been applicable to other endeavors. However, what's happening now is not the sporadic or disconnected nature of things in the past or the second wave of READER RABBIT eduware. It's bigger than that; it's multi-faceted, organized, and aimed at new targets. As Rejeski points out, "Our government spends billions each year on simulation games alone." That's just one market, and now the military is getting serious about games (AMERICA'S ARMY is the tip of the iceberg). As others have evangelized, including *Digital Game-Based Learning* author Marc Prensky, the lucrative corporate market, once stung by poor e-learning offerings during the Internet bubble, is now waking up. To put some numbers behind the corporate opportunity, according to freelance game consultant (and *Game Developer* design columnist) Noah Falstein, Shell Oil spends close to a billion dollars a year in training and corporate learning. A supermarket chain I met with, once I told them a game might cost two million dollars, replied, "That's what we spend on toilet paper."

From previous hits or misses in years past, several threads and a growing history are earning emerging recognition. We are better able to preach new applicable uses of games. The Serious Games Initiative is indicative of this, and we are working to provide improved visibility for games to be seen as tools, including ways to see more projects in the mold of SIMHEALTH, VIRTUAL U, or AMERICA'S ARMY be created. Nor are we alone. M.I.T.'s Games-to-Teach project is organizing research on an entire matrix of demonstrated learning capabilities of games. Carnegie Mellon University's Entertainment Technology Center is becoming a hotbed of applied game-technology transfer. Both the International Game Developers Association and the Interactive Digital Software Association are also making contributions, and were present at a conference on this subject last February in Washington, D.C. They know a sustained flow of such products

continued on page 71

continued from page 72

and initiatives will pay off exponentially for the industry. This mix of research and coordinated outreach, combined with other initiatives, will burst open the dam.

So what does this mean for your better future as a game developer? It means possible new avenues of revenue and a stabilizing second market to steady things amid the unstable world of game development. No doubt there will eventually be games — built and funded for "serious" purposes — achieving crossover commercial success.

What should your role be? First, you should realize that this is a different business. Serious games require adjusted practices, which could include forgoing royalty upside, publicly releasing source code, creating features for instructors as well as players, entirely different funding means (including government contracting), and looking at projects that while potentially significant, may not result in a blockbuster payday. Also, project ramp-ups may be longer, and the clients may need some "What's a game and how does it work?" hand-holding. Many projects may be smaller in revenue, but there already exist project budgets with seven figures attached to them. So if you want to participate in this new arena, bring your skills, but adapt your methods.

Finally, Serious Games needs your help. If we are to bring the full force of the game industry into new areas, be they to create simulations for policy makers and the public or to show that e-learning can be truly fun, it has to be an organized vocal effort. You can help by getting involved with projects such as ours. We are announcing a number of initiatives via SeriousGames.org, the IGDA, and elsewhere, giving you ample opportunities to contribute, and likewise benefit. The rewards, besides new revenue, just might include helping to build a better world.

BEN SAWYER | *Ben is president of Portland, Maine–based Digitalmill, Inc. He assists several "serious game" projects and spearheads outreach efforts for the Woodrow Wilson Center's Serious Games Initiative. Contact him at bsawyer@dmill.com.*

FOR MORE INFORMATION

Serious Games Initiative: www.seriousgames.org
Virtual U: www.virtual-u.org
IGDA: www.igda.com
MIT's Games-to-Teach: http://cms.mit.edu/games/education/