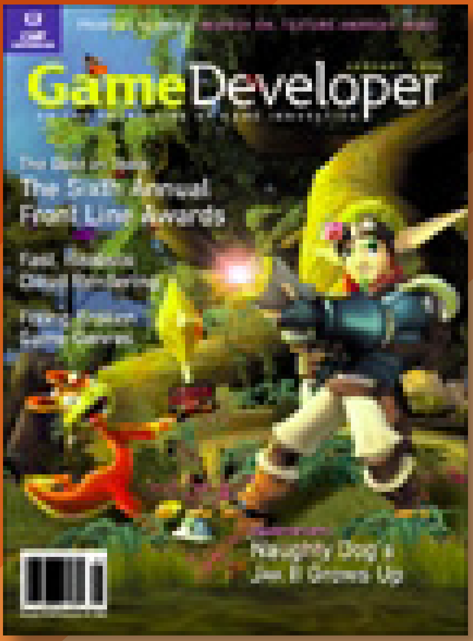
gd

GAME DEVELOPER MAGAZINE

JANUARY 2004



UBM
Tech

Return

GAME PLAN

LETTER FROM THE EDITOR

China: Over One Billion Underserved

Everybody talks about videogames going global, but when a single country such as China represents one-sixth of the world population, new forays into vast underserved markets have industrywide implications. Such is the case with both Nintendo's and Sony's entrées into the Chinese market: Nintendo has unveiled a China-specific piracy-resistant console system, while Sony has announced plans to introduce the Playstation 2 there.

Sony is launching the PS2 in China at around 1,988 yuan, or $240 (relative to per capita GDP, it's as if a PS2 cost an American $2,000). While the system is extremely expensive, the software will be much cheaper than software sold elsewhere, being subject to different market forces in China, namely those of the black market. It's well known that the piracy rate in China is near-total (94% according to Harvard's Center for International Development), compared with a relatively manageable 37% in Japan, 24% in the U.S., and 26% in the U.K.

The piracy issue is just one example of why entering, much less succeeding in, the Chinese market is more complicated than simply deciding that if every person in China bought your product for $1, you'd have $1.3 billion. For starters, two-thirds of the population still ekes out a rural existence; on the flipside, more than 60 million Chinese now sport disposable incomes greater than $10,000, a total that continues to rise rapidly.

Is Microsoft at a disadvantage by not bringing Xbox to the Chinese market at the same time as Sony and Nintendo? Yes and no. On one hand, market share is power, whatever the market, and arriving third is not a strong position. On the other hand, Microsoft, as an American company, faces steeper obstacles to trade in China than China's Japanese neighbors. If Nintendo and Sony can pave a path for a Chinese videogame market, Microsoft would have less risk to shoulder initially. However, the liabilities of entering China would remain significant for Microsoft for the foreseeable future.

The cultural oceans between China and the West are vast and deep, including a virtually nonexistent conception of intellectual property ownership such as most Westerners have. The communist government involves itself heavily in business matters, reserving — and exercising — the right to alter economic regulations drastically and without warning, which can wreak havoc on any company's long-term business strategy there. Furthermore, American companies face the unique challenge that the Chinese government keeps the yuan's valuation artificially pegged (which is to say, undervalued) relative to the now-sagging dollar. This tilts the Sino-American trade imbalance far in China's favor, evaporating the potential for U.S. companies conducting business there to profit from favorable fluctuations in an open currency exchange market. Citing loss of American jobs to artificially cheap Chinese exports, the U.S. is putting increasing pressure on China, both via direct talks and through the World Trade Organization, to revalue and ultimately float the yuan. Still, the Chinese government maintains that it cannot stabilize its currency on its own for at least 10 years.

Certainly there are western corporations that have blazed trails, persisted, and become successful in China, and certainly the gang up in Redmond are hatching some plan for the Chinese market. Then again, following Sun's "Shanghai surprise" announcement at Comdex that it had struck a deal with the Chinese government for its open-source software, Microsoft might just consider turning its attention to India's billion folks, hoping for the better.

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

EDITORIAL

Editor-in-Chief
Jennifer Olsen *jolsen@cmp.com*
Managing Editor
Jamil Moledina *jmoledina@cmp.com*
Departments Editor
Kenneth Wong *kxwong@cmp.com*
Product Review Editor
Peter Sheerin *psheerin@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*
Noah Falstein *noah@theinspiracy.com*
Steve Theodore *steve@theodox.com*
Advisory Board
Hal Barwood Designer-at-Large
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: 404.658.1415
Account Manager, Northern California & Midwest
Susan Kirby e: *skirby@cmp.com* t: 415.947.6226
Account Manager, Western Region & Asia
Craig Perreault e: *cperreault@cmp.com* t: 415.947.6223
Account Manager, Target Pavilion, Education, & Recruitment
Aaron Murawski e: *amurawski@cmp.com* t: 415.947.6227

ADVERTISING PRODUCTION

Advertising Production Coordinator Kevin Chanel
Reprints Terry Wilmot e: *twilmot@cmp.com* t: 516.562.7081

GAMA NETWORK MARKETING

Director of Marketing Michele Maguire
Senior Marcom Manager Jennifer McLean
Marketing Coordinator Scott Lyon

CIRCULATION

BPA INTERNATIONAL
Game Developer is BPA approved

Circulation Director Kevin Regan
Circulation Manager Peter Birmingham
Asst. Circulation Manager Lisa Oddo
Circulation Coordinator Jessica Ward

SUBSCRIPTION SERVICES

For information, order questions, and address changes
t: 800.250.2429 or 847.763.59581 f: 847.763.9606
e: *gamedeveloper@halldata.com*

INTERNATIONAL LICENSING INFORMATION

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

EDITORIAL FEEDBACK

editors@gdmag.com

CMP MEDIA MANAGEMENT

President & CEO Gary Marshall
Executive Vice President & CFO John Day
Executive Vice President & COO Steve Weitzner
Executive Vice President, Corporate Sales & Marketing Jeff Patterson
Chief Information Officer Mike Mikos
President, Technology Solutions Robert Faletra
President, CMP Healthcare Media Vicki Masseria
Senior Vice President, Operations Bill Amstutz
Senior Vice President, Human Resources Leah Landro
VP & General Counsel Sandra Grayson
VP, Group Publisher Applied Technologies Philip Chapnick
VP, Group Publisher InformationWeek Media Network Michael Friedenbe
VP, Group Publisher Electronics Paul Miller
VP, Group Publisher Enterprise Architecture Group Fritz Nelson
VP, Group Publisher Software Development Media Peter Westerman
VP & Director of CMP Integrated Marketing Solutions Joseph Braue
Corporate Director, Audience Development Shannon Aronson
Corporate Director, Audience Development Michael Zane
Corporate Director, Publishing Services Marie Myers

CMP United Business Media | GamaNetwork

WWW.GAMANETWORK.COM

Return

INDUSTRY WATCH

KEEPING AN EYE ON THE GAME BIZ | *kenneth wong*

Videogame middleman closes shop. Capital Entertainment Group (CEG), founded by Xbox's co-creators Seamus Blackley and Kevin Bachus, disbanded, citing a lack of funding to keep its operations going. Blackley, Bachus, and a few other videogame veterans envisioned CEG becoming the middleman between developers and publishers. As the industry's first independently funded production company, CEG planned to develop high-quality games — including original but risky projects — and then find publishers who would market, sell, and distribute the games for a share of the profit. CEG discovered its first publishing partner in Sega of America. Bachus observed that the industry will likely adopt CEG's business model in the next three years, but the challenge is "finding investors who share the appetite for risk."

Nintendo to debut new machine in Japan. Nintendo declared its plan to debut a new videogame machine in Japan next year. It did not specify whether the new machine, scheduled to be unveiled at E3 2004, will be a handheld; however, a company statement revealed that the product is not related to either the Game Boy Advance or Gamecube successor. The announcement came soon after the company posted its first-ever loss ($26.8 million, in the six months ended in September) since going public in 1962.

The motion pod for WILD EARTH, an interactive motion simulator.

Unauthorized N-Gage-ment. Nokia's mobile gaming device N-Gage, which drew criticism from the press for its high price ($299) and limited game offerings, took another blow as hackers began reporting on web sites and bulletin boards that they had cracked Nokia's proprietary N-Gage software. This raises the possibility that the games intended for the N-Gage handset can now be played on any cell phone or mobile device using the Symbian operating system, also used by N-Gage. Club-Siemens, an unofficial site run by a Norwegian duo working in London, posted screenshots of what it claims to be N-Gage games running on a Siemens AG SX1 handset, reportedly without any lag or operational deterrence.

WILD EARTH goes to theme parks. WILD EARTH, the 2003 Independent Games Festival Game of the Year, will soon be available as a motion simulator at selected zoos, amusement parks, and other entertainment environments, according to developer Super X Studios, which created the game. The adventure will allow two visitors to vicariously go on a photo-safari trek through the Serengeti. While demonstrating the technology at the International Association of Amusement Parks and Attractions (IAAPA) conference this winter, producer-developer James Thrush commented that zoos are particularly interested in the "motion pod" for its interactivity, a step beyond the current crop of passive ride films. The worldwide availability of the WILD EARTH ride is slated for early 2004.

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

THE TOOLBOX

DEVELOPMENT SOFTWARE, HARDWARE, AND OTHER STUFF

Gamepak for Truespace 6.6. Caligari recently released Gamepak, a game-development extension for its general-purpose 3D modeling software Truespace 6.6. Using Truespace's existing modeling, texturing, and animation tools as a foundation, Gamepak allows users to generate game content in file formats that are compatible with other content-creation systems. Gamepak is priced at $199. www.caligari.com

Stitch up a panorama. Realviz announced Stitcher 4.0, the latest version of its panorama-creation product. Stitcher allows users to collate multiple 2D photographs and graphics into wide-angle 3D panoramic views. Version 4.0 offers interoperability with Adobe Photoshop, interactive Quicktime VR viewing, enhanced interface and workflow management. Stitcher is available for Mac as well as PC for $499. www.realviz.com

Renderware Physics makes its debut. Criterion Software, which acquired Mathengine's physics IP, has released Renderware Physics. Available as either a stand-alone product or a component of Renderware Platform, the new physics engine features character dynamics, rigid-body dynamics, terrain collision, flexible primitives, customizable pipelines, configurable joints, multi-platform solvers, and more. Price varies depending on configuration. www.renderware.com

UPCOMING EVENTS CALENDAR

DIGITAL GAMES SUMMIT
Las Vegas Riviera Hotel
Las Vegas, Nev.
January 7, 2004
Cost: $99–$499
www.ihollywoodforum.com

INT'L CONSUMER ELECTRONICS SHOW
Las Vegas Convention Center
Las Vegas, Nev.
January 8–11, 2004
Cost: $75–$896
www.cesweb.org

Inspeck EM

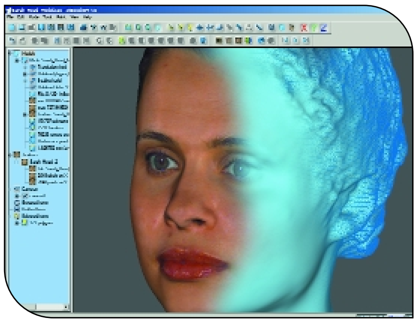
by michael dean

Inspeck EM is Inspeck's suite of editing and merging tools for manipulating 3D images of organic objects acquired with the company's line of digitizing cameras. As with traditional laser or point/grid scanning, the acquired data must be post-processed in order to allow its import into traditional modeling and animation packages, such as Max or Maya. Unlike other 3D data acquisition systems, Inspeck EM captures texture and model information concurrently, and it also successfully scans hair.

With the Inspeck system, raw data is brought in from the 3D camera/digitizer and processed in an intermediate tool called FAPS (not reviewed here), which defines the digital images as individual 3D objects and textures. These are finally brought into the EM software, which gives users the tools they need to create fully refined and functional 3D models along with correctly mapped textures.

The process begins with the model loading, and the user chooses all of the models (usually around eight or nine of them) representative of the different views and angles of the complete object, which will need to be assembled together (called "registration" in EM) to create the full model. For a human head, the user would have a face-on view, another view at 45 degrees (cheek view), another at 90 degrees (ear view), and so on, all the way around the head.

From here, it is up to the user to assist the software in determining how the models are to be merged together.


Building a 3D model of a human head from camera-acquired data in Inspeck EM.

Once the starting model is loaded, such as the face-on model, it becomes fixed in space, and the user is required to tell the software which model attaches to it and where. This is a fairly painless process which involves visually selecting arbitrary corresponding, overlapping points on each mesh. This is easily achieved, as there is an accurate, pre-mapped texture already applied and visible on each model section. Therefore, the user can select freckles or moles from each model section of a head and use these as reference points. After this is done in turn to each section of the complete object, all of the mesh objects are correctly placed in their corresponding positions to each other, and the user can merge these model sections into a single high-quality, accurate 3D model. A UV-accurate texture map is created at the same time (able to be generated both cubically and cylindrically), and with a bit of assistance from an image-editing package, such as the accurate and efficient integrated one, the 3D model is usable almost instantly.

There are, of course, unavoidable issues with the initial data that keeps the overall EM process from being completely automatic. In general, each model section has an inherent overlap associated with it (for example, the frontal model of a head contains some data from the cheek model, and vice versa). There are settings available that will automatically trim out much of this overlap, but in turn, this may cause other problems, such as holes in the mesh. It is best to keep the automatic trimming to a minimum and to delete by hand as much of this overlap as possible. This is one aspect of the software that can become very tedious, as the selection tools and the ability to add or subtract from the selection is prone to uncorrectable user error, which may require a restart of the selection process.

There are other interface issues as well. Simple viewport manipulations, which experienced users of other 3D software take for granted, are not so simple here, especially when working in registration mode. Rotations and translations are only possible in the 3D view, and not possible in the orthographic views. Zooms are achievable in the ortho views but are not simple or quick enough to be transparent in the user's workflow.

I have found that simply breezing through the registration process, almost ignoring the cleanup process, and going straight into merging the sections into one model and then exporting is the best bang-for-the-buck technique. The software is going to give you a nice model even if you don't have time to

MICHAEL DEAN | *Michael has over 10 years of experience as an artist in the game industry, and does everything in the art process from painting textures to animating characters.*

✶✶✶✶✶	excellent
✶✶✶✶	very good
✶✶✶	average
✶✶	disappointing
✶	don't bother

execute the recommended process.

However, the software's functionality doesn't end at registration, merging, and export. A primary concern of many modelers, especially game modelers, is keeping meshes lean and mean. Digitally scanned data has never met this challenge well. Usually, the process with a scanned 3D model involves a lot of optimization in another 3D package to bring the polygon counts down to a reasonable level. That's not usually so bad in and of itself; however, when the artist has, say, 20 heads to do and each of them needs to be scanned and brought down, it introduces all kinds of new problems. The optimization of several meshes, if using automatic methods, usually will vary dramatically. Vertex counts, weights, and bone influence between the models are different enough that the artist has to start the weighting process over and over again from scratch.

Luckily, the EM software contains a very nice morphing feature, which allows the user to morph from a source object (for example, an optimized, usable in-game head) to a target object (for example, a dense, scanned head). Surface topology is very well maintained within the user-set constraints, and upon import back into a 3D software package, the artist finds that the method used to optimize the original model is well applied to the morphed version of that model. Because the morph targeting is based upon point-cloud selections, there is less worry about the morphing being generic. This process is fairly painless and can significantly ease the burden of creating a large number of similar models from scanned data of very different subjects.

I highly recommend this software along with its associated sibling software and hardware for quickly and accurately producing usable scanned organic data. It takes some getting used to, and it's difficult to just dive right in and work as you always have. Once it's learned, however, I think that any artist will find that it is an efficient way of digitally capturing and processing many real-world objects into a game development workflow.

INSPECK EM ✶✶✶✶

STATS
Inspeck
Montreal, Quebec
514.284.1101
www.inspeck.com

PRICE
$3,500 (sold with digitizer)

SYSTEM REQUIREMENTS
Windows 2000/XP, Intel Pentium III 800MHz or higher, 512MB RAM, 3D accelerator.

PROS
1. Accurate organic models.
2. Adept at reproducing hair.
3. Simultaneous model/texture generation.

CONS
1. Clunky interface.
2. Tutorials need work.
3. Some stability issues.

Digital Anarchy's Texture Anarchy

by mark peasley

Texture Anarchy is a new series of filter plug-ins available for Photoshop. The set consists of three plug-ins that give the user the ability to create very sophisticated procedural textures directly on a Photoshop layer. A procedural texture is one based upon mathematical algorithms rather than real-world images, and can be used to mimic some of the patterns found in nature.

Once installed, the new plug-ins are available through the filters dialogue. They show up as Texture Anarchy Explorer, Tiling Texture Anarchy, and Edge Anarchy. All three tie into the same basic procedural texture generator, although each gives the user a different set of controls and functionality.

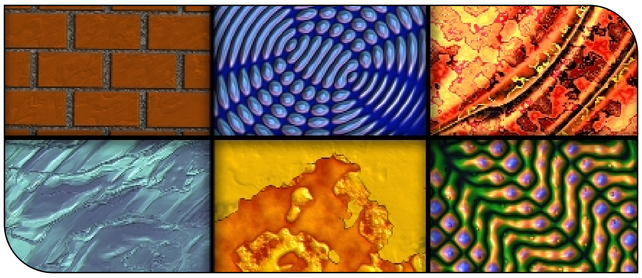
Texture Anarchy Explorer allows the user to create textures from a large selection of presets, randomly generate new textures using the mutation sliders, or control the creation "recipe" of a new texture from scratch. Tiling Texture Anarchy is almost identical to the Explorer but allows for the creation of perfectly repeating or tiled textures. The final plug-in, Edge Anarchy, is all about creating ornamental or distressed edge effects for images and text.

The plug-ins have rooms or levels, with each successive room having more in-depth control over the creation and blending of the texture. The first room that users see after opening up the Explorer or Tiling plug-in has basic editing controls such as blend type, opacity, and bump percentage for a given texture. In addition, one can control the size, rotation, pan, and on/off component of a given texture in the three main areas of color, alpha, and bump. For those who just want to experiment, there are a series of buttons that allow the user to randomize or "mutate" all of the elements that make up a texture. Through the main menu controls at the top of the screen, the user can access preferences, presets, specific output channels, and even a built-in screen saver which cycles different textures onto the screen.

The layer editor, accessed by clicking on a texture in the color or bump slot, allows one to alter how a texture composites with other textures, colors, and masks. By double-clicking on a texture in the layer editor, users are taken to the deep noise editor. From here, one can select from 38 different noise types that are the foundation of the textures, as well as alter the opacity, blend type, and color gradients.

Tiling Texture Anarchy looks and behaves almost identically to the Explorer except it outputs perfectly tiled textures. By creating a Photoshop pattern from the resulting file, any sized area can be filled with the seamless texture.

Edge Anarchy allows for the creation of decorative or distressed edge effects on images or text. A word of caution — the render mode needs to be set to "fast" or "normal" for optimal use. Setting it to "high quality" can easily lead to out-of-memory system errors on even medium-resolution images, although Digital


Images produced with Texture Anarchy.

Anarchy says that setting is for use with low-resolution images.

Overall, Texture Anarchy is a reasonably well-designed set of plug-ins. Pros include unlimited texture combinations, affordability, relatively short ramp-up time, and a thorough manual. Downsides are that the UI layout and functionality is sometimes unintuitive, the high demands Edge Anarchy places on RAM, and the somewhat limited use of procedurally generated textures in game development. Available on Mac or PC at $149, Texture Anarchy is a solid addition to your Photoshop plug-in arsenal.

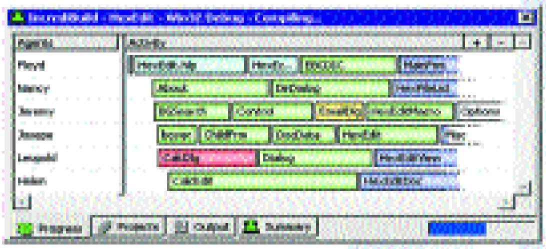
★★★½ | Texture Anarchy
Digital Anarchy
www.digitalanarchy.com

Mark creates racing games at Microsoft Game Studios.

Slow C/C++ Builds?

IncrediBuild accelerates C/C++ builds by distributing compilation tasks across the network, cutting down build time by 90% and more!

Download FREE, Fully Functional 30-Day Trial!



- Simple setup, requires no changes in code or settings
- Compatible with any Visual C++ Win32 project
- Fully integrated with MSVC's IDE

XOREAX

www.xoreax.com

The OpenGL Extensions Guide
by Eric Lengyel

reviewed by jeremy jessup

In *The OpenGL Extensions Guide*, Eric Lengyel discusses specific extensions relevant to game development. The book spans 19 chapters totaling 670 pages and retails for $59.

Extensions define new tokens and/or functions to serve as an interface to specific hardware features. While they allow for the OpenGL standard to utilize cutting-edge advances in technology, the use of extensions compromises the cross-platform operability until the extension is promoted to a core feature of the library.

In the first chapter, Lengyel describes how to query a particular OpenGL implementation for the existence of a given extension on Windows and Apple computers, covering 79 of the 338 officially recognized OpenGL extensions. The subset selected for the book is appropriate for the target audience, with many of the Unix workstation (SGI, HP) extensions omitted in favor of PC/Mac functions — in particular those created by Nvidia, which account for a third of the total.

Each extension is approached in a systematic and consistent manner beginning with a table summarizing the OpenGL version, dependencies, promotions, and related extensions. Then, the extension is concisely discussed and followed by a description of each new set of tokens or functions. The reader should have a working knowledge of OpenGL and the graphics pipeline. Unfortunately, there are no examples or sample code.

While every official extension is documented on the OpenGL web site, that information is very minimal. Lengyel does an excellent job providing a broader view of the reason and usage of an extension by presenting the functional information in a clear and concise manner. There is some overlap, but Lengyel's book is actually more accessible for the OpenGL developer than the OpenGL web site.

The last two chapters are the most exhaustive, each spanning over 100 pages. Here, Lengyel describes the vertex and pixel shader language in OpenGL. He also covers the grammar, syntax, constraints, registers, and sub-instructions of the Architecture Review Board and Nvidia vertex program extensions.

It would have been helpful to have a hypertext version of the book to ease cross-referencing various symbols. Also, the lack of example code or any experience-based guidance makes it inappropriate for beginners, as you cannot learn OpenGL from the book. Despite these shortcomings, the book does provide a valuable reference of many of the key graphical extensions for the OpenGL developer. For those developers frequently using OpenGL extensions, this is a great desktop reference that brings together scattered information in a clear manner.

★★★½ | *The OpenGL Extensions Guide*
Charles River Media
www.charlesriver.com

Jeremy is a programmer for Rockstar San Diego.

Return

Game Designers Without Borders

Nintendo's Shigeki Yamashiro on going global

Shigeki Yamashiro is in a unique position in the game industry. Having learned the art of game development from none other than Shigeru Miyamoto, he now applies that knowledge as president and producer for Redmond-based Nintendo Software Technology (NST). He started 15 years ago at Kyoto-based Nintendo Co. Ltd. (NCL) working for Miyamoto on F-ZERO, and was also involved with DONKEY KONG COUNTRY, PILOT WINGS 64, and KEN GRIFFEY JR. BASEBALL. After moving over to NST, he produced WAVE RACE: BLUE STORM and 1080°: AVALANCHE.

Shigeki Yamashiro wants to make games for everyone — literally.

Running NST gives Yamashiro access to diverse resources. In the middle of a project, he can request NCL to give comments on the latest build, and Miyamoto and his producers give him rapid feedback. Furthermore, he has access to Nintendo of America's (NOA) U.S. market information and their evaluation and testing team. On top of that, the roster of game developers at NST sounds like the United Nations of game design, with Nigerian-born, Spanish-raised lead designer Vivek Melwani and a team from all over Europe. Yet the team wasn't assembled just for diversity's sake — according to Yamashiro, "this is the natural result of bringing the right people together."

Still, we were curious how Yamashiro faces the challenges of integrating such a team while applying the craft, so we asked him to share a few details about his methods.

Game Developer: What lessons did you learn working with Shigeru Miyamoto that you found repeatedly applicable down the road?

Shigeki Yamashiro: I had an opportunity to work with Mr. Miyamoto for seven years at NCL. I learned a lot from him, but one example I can give you is that you always need to maintain the player's perspective. Is it easy enough to understand? Fun to control? Interesting enough to buy? These types of questions should not only be asked but should be part of the overall project goal.

GD: What processes did Miyamoto teach you to maintain the player's perspective?

SY: Frequent playtesting is a basic requirement that Mr. Miyamoto expects from all game designers who work with him. While the game designer is building the game design, you need to review your design from the player's point of view. Of course during the later stages of game development, when your game is playable, you can ask someone to play your game while you watch from behind the player. Using this technique you will see a lot of issues that should be addressed.

GD: How would you compare game development methods between NCL and NST?

SY: NCL has been developing games for many years and, because of this, employees there are very familiar with game development. Here at NST, we are trying hard to perfect our own unique method of game development — using lessons from both Japan and here in the U.S. Other differences come from cultural background, and that makes for hot discussions about game design direction from a variety of perspectives. But I don't see any differences regarding enthusiasm for making videogames.

GD: How do you integrate Nintendo's and international developers' techniques?

SY: A typical western developer uses a design document and makes a videogame by following that document. While making 1080°: AVALANCHE we didn't create a strict game design document at the beginning. We had key concepts that we wanted included in the game but many of the details and design changes were implemented during the development process. We need that flexibility because our primary focus is on play control and interactivity for the player. For example, implementing the feeling of speed or the rush of an avalanche chasing you is easy to conceptualize, but it's tough to write into a document exactly how to achieve those sensations.

GD: What is the main advantage of developing in the U.S.?

SY: NST's ultimate goal is making a videogame that appeals to the international market. POKÉMON is a worldwide success because the style of play appeals to game fans of all nationalities. Since NST is a new company, we are still working out the formula for making games that are accepted worldwide. To begin we decided to focus on the U.S. market, and once we make a videogame that is successful with the U.S. audience, we can step up and challenge our ability to entertain the world.

GD: To what extent will NST be involved in developing games for the new Nintendo device reported to be released next year?

SY: At NST, we are always looking to find a new game experience that people will love. I can't tell you what we are doing now, but don't forget we were one of the initial developers for Nintendo Gamecube.

GD: What games are you playing now?

SY: MARIO KART: DOUBLE DASH!! and ANIMAL CROSSING E+ (Japanese version).

Return

Designing the Language Lerp

During the autumn of 1994, I attended a small conference in New Mexico called the USENIX Symposium on Very High Level Languages. I was in college at the time and also a member of the contingent that thought "very high level" meant features like closures and continuations. The conference had been organized rather by folks who thought "very high level" meant "language features that help you get a lot of work done quickly."

Among many things that happened that weekend, I had a conversation with Larry Wall, the creator of the programming language Perl, which had already proven to be quite popular. During this discussion I suggested that Perl was icky because it wasn't very "orthogonal."

In language design, orthogonality means a lack of redundancy in the language's feature set; there's only one fundamental way to accomplish any particular kind of task, so ideally you achieve maximum expressive power with minimal linguistic complexity. Among academic language designers, orthogonality is one of the main sources of beauty in language design.

Wall took the position that orthogonality isn't actually a good idea. He explained, if you want to travel from point A to point B in the example here (Figure 1), you'll have to travel farther if you can only move along the orthogonal grid lines (you must travel the Manhattan distance); what you really want to do here is move directly along the diagonal line between A and B (and you must only travel the L2 distance).

Back then, I regarded this diagonality analogy as somewhat ill-formed, an overzealous application of the term "orthogonal." But now, years of real-world experience later, I see that Wall is right. Also, I am now more comfortable with mathematics, and I no longer think of the diagonality as an ill-fitting metaphor. Rather, I think it's a fairly accurate description, perhaps even an isomorphism rather than an analogy. Nowadays you can find references online to Wall's philosophy, crediting him for designing Perl to be a "diagonal language."

Perl's Doing Something Right

If you're fluent in Perl and working on a problem in Perl's domain (string and file processing), the language offers a huge productivity boost above its competitors. Part of the reason for this effectiveness is the diagonality principle. But it's not the only reason — some things can be attributed to the language's general high-levelness and the availability of many utility functions. Also, when programming in Perl, one tends to adopt an attitude that software engineering and meditating on structure are not so important to the task at hand. This allows one to progress quickly toward a solution, unfettered by design concerns. Though the latter effect is separate from the diagonality principle, I think diagonality encourages it.

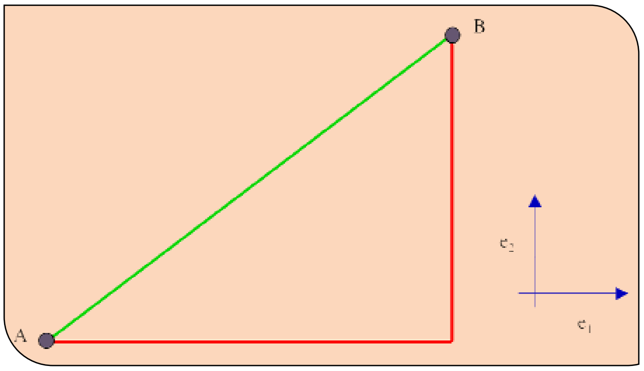


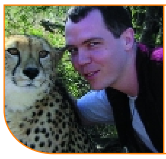
FIGURE 1. An illustration of straight-line interpolation versus traveling along orthogonal vectors to get from A to B, with e_1 and e_2 as the basis vectors of the space. The green hypotenuse of the triangle is the short path; the red sides of the triangle are the Manhattan path.

Last month I discussed some of the frustrations we face with regard to software development. As a thrust toward some kind of solution, I've decided to make a new language, adopting Perl's diagonality principle but taking it in a different direction. Whereas Perl originated in the system administration world, I want my new language to be a tool specifically for writing gameplay code, scripted events, and other manipulations of objects in a world.

I'm whimsically naming my new language "Lerp" for two reasons: the name is a permutation of "Perl," and a common abbreviation for "linearly interpolate." Going back to our get-from-A-to-B-on-a-grid example, Lerping from A to B gives you the shortest-line path (assuming, ahem, that the space is linear).

Basic Design of Lerp

Lerp is a fusion between the imperative style of a language like C, and the declarative style of a language like Prolog. When I read introductory books about Prolog, I am always struck by how simple, intuitive, and powerful the language seems, as long as I am still in the early chapters of the book. But before long, say, around Chapter 3 or 4, Prolog suddenly gets



JONATHAN BLOW | *Jonathan (jon@number-none.com) has been in New York City for 10 days now and he still hasn't seen the Statue of Liberty.*

LISTING 1. QUERY, TUPLE RESULTS

```
proc show_sisters(Database db) {
    results = db.[´sister ?x ?sis];

    each results {
        print("A solution is: ", $_, ".\n");
    }
}
```

ugly. The reason is the orthogonality principle — Prolog has some cool ideas about pattern matching, but then the language designer had to go and try to build an entire Turing-complete language out of those few ideas. As the old saying goes, "When all you've got is a hammer, everything looks like a nail." The early intuitiveness of Prolog quickly falls prey to the weird quasi-declarative semantics required to accomplish imperative-style tasks.

I intend to make the pattern-matching part of Lerp more intuitive. I'll be using a beefed up version of last month's simple predicate logic system, integrated into the language in the same way Perl uses regular expression matching: you have some imperative code that goes along statement by statement; then one of the statements happens to be written in a declarative style, indicating a pattern-matching operation. The results of that pattern match are subsequently available as variables in the imperative code. Let's say that I want to write a simple line of Perl for extracting hours, minutes, and seconds. The input is a variable called `time`, which is expected to contain a string formatted "`HH:MM:SS`" (where H, M, and S are digits). The `=~` is a pattern matching operator, and the sequence to the right of it is a declarative description of how the string should be parsed. So the code will look like this:

```
($hours, $minutes, $seconds) = ($time =~ /(\d\d):(\d\d):(\d\d)/);
```

The regular expression facilities in Perl are responsible for a significant chunk of Perl's power. It's my hope that the predicate logic system can play a similar role in Lerp. Instead of pattern matching on strings, I want to match patterns about facts in the game world.

Integrating Pattern Matching

I want the pattern matching to integrate seamlessly into the language design. I decided early on that a pattern-match operation would return a data structure that could then be iterated through, and made it a goal for the language to possess powerful iteration capabilities.

Recall that last month I used parentheses and unquoted identifiers in order to designate a query: `(sister ?x ?y)`. This query was then matched against a database of fact or of inference rules that were used to create facts.

Last month's syntax was in keeping with a tradition for implementing simple predicate logic interpreters in LISP-like languages. But now that I want to embed pieces of predicate logic in an imperative language, I find that I would rather use parentheses for the traditional role of grouping imperative expressions, and I'd like unquoted identifiers to indicate imperative variables. So I adopted a new syntax where a query uses square brackets, and predicate identifiers use a single quote: `[´sister ?x ?y]`.

I could set up the language to use one global database to represent "the game world" but it seems best to provide the ability to instantiate arbitrary databases. I will treat these databases conceptually like C++ classes, and I will borrow the dot operator "." to indicate a database query. Finally, I provide the "`each`" keyword as a simple way of indicating an iteration, with `$_` being an implicit variable that gets assigned the value of each iterated item, as in Perl. A simple program to print out all the solutions of `[´sister ?x ?y]` looks like Listing 1. It will print out some results that look like this:

```
A solution is: [sister mark ann].
A solution is: [sister mark mary].
```

LISTING 2. QUERY, NAMED RESULTS

```
each db.[´sister ?x ?sis] {
    print(sis, " is a sister of ", x, ".\n");
}
```

So far we can perform a query, but this doesn't seem very integrated or powerful; the situation just doesn't have that Perl magic to it yet. As the return value of that query operation, I just got some tuples back. If I want to dig inside the tuples, I will need to perform some extra operations to extract things. This would be a bit tedious, and it seems like the most common usage case (for example, I want to call a function, passing just the values that match the query variable `?sis` — the people who are sisters of someone).

To make this situation nicer, I added some extra power to the "`each`" iterator. If the list argument to "`each`" is a database query, the parser inspects the query to find all the argument-matching slots (like `?x` and `?sis` in Listing 1). Then it defines local imperative variables, inside the body of the iteration, for each of those slots. Now we're really in the neighborhood of non-orthogonal language constructs. The results help make things concise and easy to read. See Listing 2, which generates output like this:

```
ann is a sister of mark.
mary is a sister of mark.
```

Further Features

Often we want to perform queries on a relation that is supposed to be one-to-one or one-to-zero. For example, the `carried_by` relation (an entity can't be carried by two people — only by one or nobody). In this case it would be cumbersome to perform an iteration or to get the query result back as a tuple, so I provide a special ?? variable that means, "return the value of the item in this slot, not the whole tuple." Thus we can write this as follows: `carried_by = db.[´carrying ?? item];`

Then the value of `carried_by` will be either the guy who is carrying the item, or a null-like value if nobody is. (If multiple matches are found for `[´carrying ?? item]`, an error is thrown.)

By giving an expression as the second argument of "`each`" instead of a code block, we can make a new list out of the query results, like this: `results = each db.[´sister mark ?sis] sis;`

This makes a list containing each value of `sis` — much like the LISP function `mapcar`. I also added the ability to "lift" an iteration up through a function call. Assuming `x` is a variable containing a list, I can say this: `f(each x);` And that is shorthand for: `each x f($_);`

This is not a big gain so far, but it allows easy expression of nested iterations. Suppose `y` and `z` are two more lists, then `f(each x, each y, each z);` is shorthand for:

```
each x {
    item_x = $_;
    each y {
        item_y = $_;
        each z {
            item_z = $_;
            f(item_x, item_y, item_z);
        }
    }
}
```

except that in the shorthand version, it is assumed we don't care about the order in which the iterations happen.

Unified Data Structures

I now have this generalized database mechanism, but I don't yet have basic language support for traditional data structures (lists, trees, C++-style classes full of named slots, and the like). Rather than build those data structures as separate language subsystems, I chose to unify data handling within the language and use the database model for everything. This helps to ensure that we can leverage the expressive power of the pattern matching on all of the frequently used language constructs.

Hash table functionality, or the ability to store values indexed by arbitrary keys, is already embodied by the semantics of the database. Linked lists can be expressed within a database pretty simply, as can arrays, so I won't dwell on those. Once we have C++-style classes (coming in a few paragraphs), you could choose to build lists out of those as well.

In general, though, I doubt that structures like lists and trees will be used very often in Lerp. Usually when we implement these data structures in other languages, we're participating in a mentality of computation scarcity. We use lists because appending to them is fast, and removing an object out of the middle can be fast if the list is designed for it. Or we use trees so that we can quickly maintain a sorted collection of objects, which remains sorted as the collection is modified. We're only maintaining a sorted collection because we don't want to re-sort all the objects when it's time to use them (since that would take CPU cycles).

This CPU-scarcity attitude costs us a huge amount in terms of software complexity. Whenever it's possible not to treat computation as scarce, we can achieve much simpler, more powerful programs. These days, that's possible in an increasing number of contexts, because computers are so fast, and large portions of our programs are not speed-sensitive. That's the area I'm aiming for with Lerp. If you want to try to engineer a Lerp program to use minimal amounts of CPU, you can do that, but the language isn't designed to make it a primary concern.

Struct Something

C++ gives us the ability to define classes with named slots; I wanted a similar capability in Lerp. I decided to use the C "`struct`" keyword and to implement classes as databases. A definition that looks like this:

```
struct Something {
       Integer x = 0;
       Integer y = 1;
       String z = "Hello, Sailor!";
};
```

turns into a database with the following entries:

```
[_member x Integer 0]
[_member y Integer 1]
[_member z String "Hello, Sailor!"]
```

You can use the dot operator to de-reference struct members as lvalues or rvalues, and these are translated into the appropriate database queries and assignments. For example, "`thing.x`" is equivalent to "`thing.[_member x ? ??]`" (recall that a pair of question marks indicates that we wish to return that slot as the value of this expression; the single question mark indicates a value that we don't care about, without even bothering to name it). Under this scheme, we get introspection naturally; to find out what members are declared on a class, you can do this:

```
each thing.[´_member ?name ?type ?value] { // Do something...
```

Sample Code and Next Month

So far I have trotted out a bunch of language ideas. I'm not doing this to convince you that this particular set of language semantics is the greatest ever; rather, I want to provide lots of concrete examples of diagonal language features for games, sowing the field for discussion. Perhaps you think some of these features should work differently, or that some of them should be thrown out and replaced with entirely new ideas. That's good.

This month's sample code (available at www.gdmag.com) implements the features discussed in this article, and more. You can play with the sample programs and see how they go. Next month we'll look at some even crazier high-level features — note that we haven't done anything special to accommodate inference rules yet.

Subdivide and Conquer

Subdivision surface tools are the big growth area in modeling. In addition to increasing support from the major vendors, this year has seen one stand-alone subdivision modeler (Silo) released and another (Modo) on the way. Subdivisions have largely replaced NURBS as the medium of choice in film and visual-effect applications. After all, what's not to like? Combining the smoothness of NURBS with the ease of free-form polygon modeling, subdivisions seem to offer the best of both worlds.

Unfortunately, information on how to work with subdivisions is hard to find. This has more to do with the way subdivision products have been grafted onto older technologies than with the technology itself. A lot of the experienced subdivision modelers cut their teeth on relatively obscure programs such as Mirai or Wings3D; sometimes they appear to be speaking a dialect that's almost incomprehensible to artists with mainstream NURBS or polygon modeling backgrounds. In this and future columns, we're going to take a look at subdivision modeling fundamentals and try to clear up some of the confusion.

The algorithms for subdivision surfaces have been around almost as long as those for NURBS, but the technology has only gained prominence in the last five or six years. Pixar liberated subdivisions from academic obscurity with the release of *Geri's Game* in 1997, which spotlighted a main character modeled in subdivisions. Since then, subdivisions have rapidly become the method of choice for organic character modeling. NURBS remain more popular for mechanical and architectural subjects, although this may have more to do with the comparative maturity of NURBS tools and the misconceptions about how to handle subdivisions than with subdivisions' technological limitations.

There are actually quite a few different algorithms for generating subdivision surfaces, including the poetic Butterfly method and the Loop method (which has nothing to do with looping, but was invented by someone named Loop). The most popular is the Catmull-Clark method, developed in the late 1970s by Ed Catmull, when he was still a graduate student and not yet president of Pixar. For the remainder of this discussion, I'll be describing Catmull-Clark subdivisions, since they are the ones most game artists have seen and used.

The obvious advantage that subdivisions offer over NURBS is in the construction of the control mesh. Because most packages don't allow you to render the control network of NURBS patches, it's easy to miss the fact that CVs and hulls are just the vertices and edges of a quad poly mesh (see Figure 1). A NURBS surface can only be built out of quads. That's just what a NURBS hull is, but it's a mesh with severe restrictions. NURBS have to be built out of a regular grid of quads. You can see this quite clearly in the classic NURBS tools; bi-rails, extrusions, and lofts are all different methods of generating four-sided shapes. Even a NURBS sphere (or any revolved surface, for that matter) is actually made from a four-sided patch — it's just that its top and bottom edges are scaled down to zero length, as can be plainly seen in the texture maps on the sphere in Figure 1. The quad patch restriction means complex forms have to be built out of a network of separate patches. Unfortunately, the surface is continuous (that is, smooth) only within patches. If two adjacent NURBS surfaces aren't aligned properly, cracks or creases occur. In the last few years, tools like Maya's Global Stitch or XSI's Continuity Manager have made it somewhat easier to manage continuity between adjacent patches; nevertheless, the task is still, to put it mildly (this is a family magazine), an impediment to the natural flow of artistic genius.

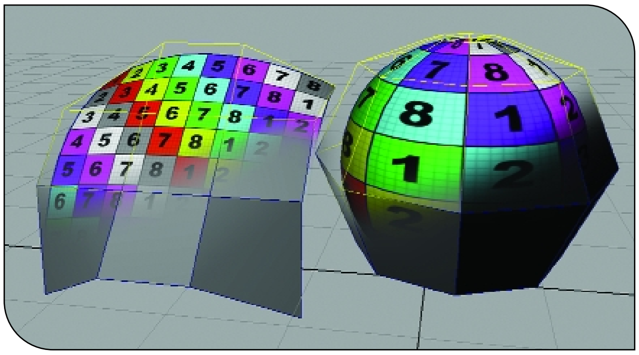


FIGURE 1. The control hull of a NURBS surface is really a quad mesh.

Subdivisions, on the other hand, can handle any arbitrary control mesh. Well, almost any — subdivision meshes have to be manifold, meaning an edge can be shared by only two adjacent faces. This isn't much of a limitation, although, in some packages, trying to perform a subdivision on a non-manifold mesh will simply reboot your machine. (If you 3D tools folks are reading this, in future releases, a simple warning dialog will be just fine, thanks.) The freeform control mesh is the reason most packages lump subdivision and polygon modeling tools together — tools for adding edges, deleting vertices, and so forth are already part of the standard polygon toolbox. Under the hood,

STEVE THEODORE I *Steve started animating on a text-only mainframe renderer and then moved on to work on games such as* HALF-LIFE *and* COUNTER-STRIKE. *He can be reached at steve@theodox.com.*

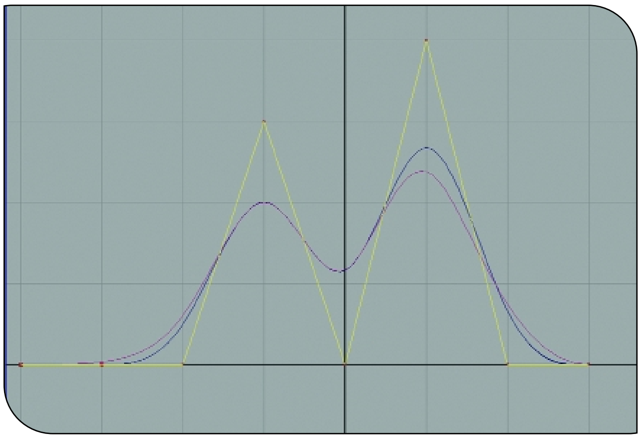


FIGURE 2. A NURBS spline (red) and a uniform bi-cubic spline (blue) with the same control hull (yellow) produce similar curves.

though, the creation of the smooth surface, whether by NURBS or subdivisions, is accomplished by recursively subdividing and smoothing the control mesh. In theory, this goes on infinitely, until the result is a theoretically perfect "limit surface." In practice, the subdivision will stop when the user (or the renderer) decides there are enough polygons and calls it quits.

Not All Splines Are Created Equal

So, are subdivisions just NURBS with a freeform control mesh? Not exactly. The difference lies in the math that handles the subdividing and smoothing processes. The difference between NURBS and subdivisions is the difference between non-uniform rational B-splines (which is where the acronym NURBS comes from) and uniform bi-cubic B-splines (maybe we should call them UBBS?). This sounds more difficult to understand than it really is. In Figure 2, we have a NURBS curve and a uniform bi-cubic B-spline curve, both generated by subdividing and smoothing the line segments connecting the control points. In both cases, any given point on the curve is affected by four control points. But as you can see, identical control meshes don't produce identical curves. We'll gloss over the mathematical reasons; for our purposes, we can put the difference as follows: in a NURBS curve, the influences of the control points are re-weighted, effectively making the curve stiffer (as you can see clearly in the illustration). NURBS curves are "normalized" and "rationalized" so they can be used to represent mathematically exact circles and ellipses; subdivision B-splines, however, always have a tiny fudge factor when trying to reproduce perfect shapes. For some CAD modelers and engineers, this difference may be critical, but for game applications, the imprecision of subdivisions is a small price to pay for greater freedom.

The difference between the two types of splines often unnerves NURBS veterans when they experiment with subdivision models. While the mechanism of pushing control points around is the same, the results of a given control input are different — and different in a squishy, nonlinear way that can be irritating, without being obvious enough to spot if you don't know what to look for. Moreover, because subdivisions are "bendier," some kinds of modeling become harder. In particular it gets harder to lock down the tangents on a subdivision surface, thus compound curves, fillets, and other mechanical shapes modeled with subdivisions often seem to lack the crispness associated with NURBS models. This isn't to say subdivisions aren't capable of representing non-organic models, but it does suggest a lot of modelers who've moved on to subdivisions have to watch out for vestigial NURBS instincts, which are just a bit out of step with the technology.

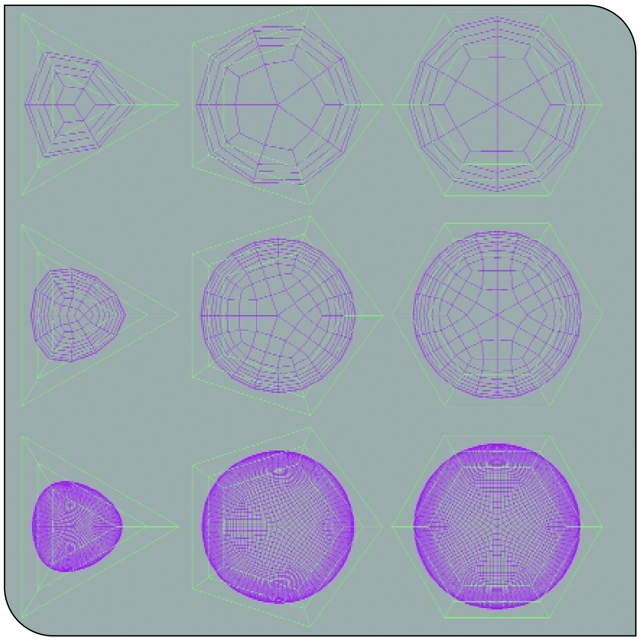


FIGURE 3. N-gons are turned into quads by subdivision, but the midpoint of an N-gon leaves an extraordinary point, with its shading glitch.

Extraordinary Points

Now, we just finished explaining that subdivisions don't have to be quad patches, right? Absolutely — you can throw a mesh made of any combination of N-gons at the subdivision algorithm and you'll get a smooth surface. However, there's smooth, and then there's *smooth*; if you want *smooth*, you have to understand a little about how the subdivision algorithm works. The first thing to know is the subdivided mesh is all quads, regardless of the topology of the original faces in the control mesh. So the triangles and N-gons will be homogenized down to quads once the subdivision begins. However, any vertex in the control mesh that sits at the intersection of more or fewer than four faces will still show up in the subdivided mesh with the same number of neighbors (because the subdivision happens exclusively inside the faces). Also, any N-gon that gets turned into quads will end up with a point at its center that has

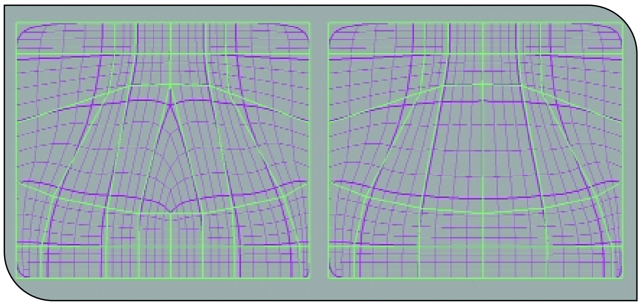


FIGURE 4. Vertices with more or fewer than four incoming edges terminate edge loops. Compare the smooth splines in the right image with the cusps created by the extraordinary points in the left image.

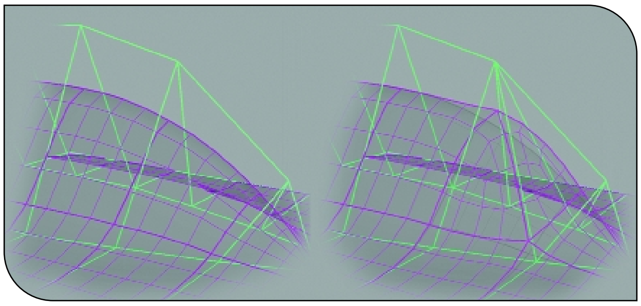


FIGURE 5. Adding edges to create an extraordinary point in the example at right interrupts the original smooth contour. Vertex positions are the same in both images.

N neighbors (Figure 3). Since the smoothing algorithm assumes quads (after all, at every recursion other than the first, it's only working on quads), it knows how to build four-way intersections with absolutely correct normals. However the normals at the intersection of more or fewer than four quads will be slightly off, meaning there will be a minor shading artifact on each non-four-way intersection and in the center of each N-gon.

Avoiding these oddball vertices (mathematicians diplomatically refer to them as "extraordinary points" or "poles") is a fetish in subdivision modeling. Some modelers argue that you should build your mesh entirely out of quads to avoid the extraordinary points, which will crop up in the middle of N-gon faces. In practice, though, this isn't always worth worrying about, since the error is going to scale down as the surrounding polygons get smaller with each subdivision. Moreover, in games we're so used to crappy Gouraud shading that it's unlikely a small normal error in a 3,000-polygon character mesh will attract much attention. On the other hand, if you are creating a 100,000-polygon shiny marble vase with lots of traveling specular highlights for a movie, you'll want to be sure the extraordinary points aren't positioned in prominent places.

Edge Loops

The real problem with poles isn't that they create shading artifacts — it's that they can change the behavior of the surface under the control mesh. If you follow discussions of subdivision modeling on the web, you'll probably notice that the term "edge loops" gets tossed around a lot (often with little or no explanation as to what it means). The simplest definition for an edge loop is a series of continuous edges that pass directly through four-way intersections in the control mesh. Edge loops are important for a simple reason: a series of edges that meets the definition of an edge loop is also the control hull of one of those B-splines we discussed earlier. Edge loops terminate at non-four-way intersections or the edge of a mesh (or they meet up with themselves and become "loops" for real). The four-way intersections are important because any other kind of intersection offers no automatic way to choose which edges continue the spline. The angle at which the edge loop passes through the vertex doesn't matter, so a T-intersection with incoming edges that look like a straight line ends an edge loop as completely as a five-way intersection that looks like a broken windshield (Figure 4).

Because edge loops generate splines, the surface under a series of edge loops is guaranteed to flow smoothly. But, as you know from NURBS modeling, the endpoints of splines (NURBS as well as subdivision B-splines) behave differently from midpoints. Thus the surface under a pole vertex will have some kind of deformity in it. You can see the effect very clearly in the view shown in Figure 5. Since extraordinary points are influenced by all of their incoming splines, they won't show up as sharp corners or points but merely as imperfections. However, if you leave an extraordinary point on a contour that's supposed to be smooth, you'll get an unsightly bulge or ding. Moreover, while the surface flow under edge loops is typically parallel to the edges in the controls, cusps are displaced in the direction of the center of all the connected vertices (Figure 4 again). In low-resolution meshes, this causes shading artifacts by disturbing the grain of the final mesh.

Does this mean you should never allow extraordinary points into your mesh? There are certainly folks on the web who advocate that, but in practice, that's impossible. In fact, you need pole points to terminate edge loops in places where you don't want clear continuity. Even more important, it's impossible to get from areas of high detail (say, the face) to areas of low detail (the back of the head) without creating poles. Generally speaking, you can hide poles either inside creases or in areas of low curvature with relatively little chance of being spotted. The only definite rule is not to place a pole along a contour that will be visible in profile.

Coming Up

That's pretty much it for the basic vocabulary of subdivision modeling. We'll return to the "grammar" of subdivisions in a couple of months, with some specific rules for quadding meshes, managing details, and integrating NURBS tools into subdivision modeling. Next month we'll clean the palate with some character rigging.

Color Sound
Synaesthesia in Games

Kandinsky had it, as did Scriabin and Messian. For Tetsuya Mizuguchi, it provided the concept behind the game REZ, while Jeff Minter's 20 years of experimentation with the idea sees him now working on UNITY. Its existence may be controversial and the spelling difficult, but synaesthesia is a concept capable of providing a fertile ground of inspiration for game designers.

Synaesthesia is defined as cross-modal perception, the ability to interpret one sensory input by another. It is most commonly understood as visualizing sound, particularly in terms of abstract shapes and colors, although it can refer to the interpretation of any one sense by another.

Unfortunately, the very evidence that suggests the condition actually exists also indicates that it is entirely specific to the person. John Harrison's fascinating book *Synaesthesia: The Strangest Thing* (Oxford University Press, 2001) provides compelling evidence that some people do indeed see specific colors when looking at words or listening to sounds, but each synaesthetic responds in quite different ways. But this column is about game audio, so how can synaesthesia be relevant for a broader audience?

Applied theory. As a digital medium, one of the unique aspects of games is that all content is stored in exactly the same way, as binary data. Only when the data needs to be displayed is it then interpreted back into a meaningful form, in the case of sound by passing it through an A/D converter. If synaesthesia in its truest sense is too specific to the individual to be practically applicable, much can still be learned from the concept through the creation of synaesthetic-like effects by building links between different sets of game data.

There are some very good instances of these ideas being put into practice. United Game Artists' REZ may not have gotten the recognition it deserves, but the level of consideration given to the combined effect of sound and imagery is impressive. During development, the graphics and sounds were created alongside each other, a rare ideal that helped to create an unusually cohesive game. Nana On-Sha's VIB RIBBON pushed this idea one stage further by using the music to help generate the game world, with the amplitude of the music tracks affecting the way in which the landscape is generated.

Modeling particle animation to sound can create highly integrated effects for games like MEDAL OF HONOR: RISING SUN.

Going mainstream. There will always be more potential in creating closer integration between sound and graphics in games that place unusual emphasis on the audio, but the synaesthetic approach can have wider application.

In games that aren't aiming for film-style realism, the music could be controlled by the physics of the environment. Abstract games such as SUPER MONKEY BALL could, like REZ, use the achievement of targets as a way to initiate new musical elements. On simpler levels where the task is straightforward, the music may consist mostly of backing tracks, while on more complex levels, where there may be a series of goals to achieve, each goal may trigger or alter a music line.

For games that use physics as an integral part of the environment, certain values from the physics engine could be passed to the music engine, making changes in the sound design. In a dark, DOOM-style game, bumping into a metal chain hanging from the ceiling could play a sinister chord, the pitch of which lowers in direct relation to the chain swinging.

Messian had visions of color when hearing sounds, while Kandinsky "heard" sound from his pictures. Similarly sound can be data driven, or more rarely, as in the case of VIB RIBBON, data may instead be sound driven.

Personalizing explosions. To take the example of an FPS such as MEDAL OF HONOR, a grenade explodes, triggering an explode animation while simultaneously playing one of a number of grenade explosion sounds. Another view might be to first randomly choose the sound when an explosion routine is called, then model the particle animation of the explosion on the sound parameters. The amplitude or frequency content of the sound could be linked to the animation properties, still allowing for some random features, creating an effect with a high degree of integration.

Synaesthetic-like ideas see the audio as much from a design perspective as an aesthetic one. Rather than treating the sound design as an extraneous element that sits on top of the game, the ideal for interactive media can be to have the audio as closely and deeply integrated into the game as possible, its creation arising out of close collaboration between the designer of the game, the programmers, and the sound designers.

PAUL WEIR | *When Paul's not off tripping the light fantastic, he runs the sound design company Earcom (www.earcom.net). Recent games include* GHOST MASTER, ROGUE OPS, FREESTYLE METAL X *and* WARRIOR KINGS: BATTLES.

Food for Thought

This is my third food-related column in a row — and the last for some time (I have to stop writing while I'm hungry). But before I move on, I'd like to share some similarities between game design and cooking.

One of the interesting qualities of game design is that it's a rare discipline that combines the logical reasoning of the left brain with the intuitive and artistic sensibilities of the right. On the one hand are the logical and scientific elements of programming, math, and physics; on the other hand are the aesthetic and softer qualities of art, music, and social interaction.

Architecture is another one of these disciplines, involving both science and art. Accordingly, *A Pattern Language* (Oxford University Press, 1977) by Christopher Alexander, an insightful book on architectural planning and building based on natural considerations, has captivated many designers and programmers.

Cooking and game development share some interesting parallels. Consider a recipe as the analogy for a game design document. Both are essentially instructions for creating a product. Both describe ingredients and how they are put together. Both require a mix of systematic steps and improvisation. And the measure of success is subjective — one person's delicacy is another's disaster.

My interest was piqued when I realized that the 400 Project rules I write about here are in many ways similar to cooking techniques. For example, one recent rule was "Provide a Single Consistent Vision." This applies well to cooking — the culinary version is "Too Many Cooks Spoil the Broth." In a great restaurant, you often find a whole army of people contributing to the preparation of delightful meals, not unlike what happens in well-run game companies or successful movie studios. Cooks also gain experience in special techniques like swiftly whisking oil into egg yolks to form mayonnaise, which applies specifically to certain other sauces (emulsions to left-brained chemically inclined chefs). This has parallels to some of the 400 rules that also apply only to distinct subsets of games.

Cooking up a hit. What is the practical value of the analogy? It can help explain some design issues to those unfamiliar with game development. Sometimes when a recipe turns out to be impractical, you just have to throw it out and start all over (no amount of tinkering will make that fallen soufflé rise). That's a useful example many successful games have followed (and some disastrous ones have ignored). I recently heard Will Wright say he threw out the interface for THE SIMS and started fresh 10 times. And Sid Meier tried three different recipes for a dinosaur game before deciding it just wasn't fun.

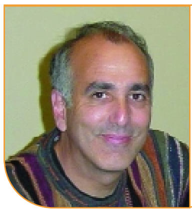
The measure of success for a game or a meal is subjective — one person's delicacy is another's disaster.

Also, one should be careful about applying a feature to a game just because it was fun in another. Lots of people love chocolate for dessert, but it doesn't belong in beef stew. A great FPS level design technique may not suit an educational strategy game, but, as with cooking, it can pay off to experiment by trying different fusion styles, applying French techniques to Asian ingredients, for example. When war-game techniques were crossbred with SIMCITY-style building, a whole new genre of RTS games took off.

From the mailbox. Finally, a little reader e-mail. Apropos to my "Too Many Cooks" comment, Gregor Koomey of The Crazy Factory points out the interesting interplay between having one single vision and getting input from many sources. It's certainly true that many great games reflect both a single vision and the incorporation of creative input from many people.

I've also had some interesting exchanges with Aubrey Hesselgren about the issue of fairness. Like consistency, it is an essential ingredient in many games but can ruin them if overdone (like using too much salt in your meals). One can be fair or unfair to the player in many ways. Should a game's AI use only the information available to a human opponent, or should it cheat by using hidden information? It would seem fair to avoid cheating, but it is hard to construct AI that even approaches human intelligence in limited settings, so it may be much more enjoyable for the player to face a challenging opponent. This can be done by judiciously giving the AI some hidden information, or letting it take advantage of instantaneous computer-driven reflexes, and then adding some random factors for disguise.

If the game's AI is made more enjoyable by being objectively unfair to the player, does the overall game become subjectively more enjoyable? There are some interesting rules and trumping information to consider.



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.*

FRONT LINE AWARDS

Illustration by Jason Howard Statts

GameDeveloper

2003 Front Line Awards

For *Game Developer*'s 2003 Front Line Awards, the editors and judges must have discovered the universal question of life, the universe, and everthing in the pursuit of the ultimate hardware and software tools for game development, for we came up with exactly 42 finalists in our Programming, Art, Audio, Hardware, and Game Components categories.

These finalists were culled from nominations from the public and our judges, and all of them had to have been released between September 1, 2002, and August 31, 2003. These 42 represent those tools that our judges deemed the best, most innovative, or most useful products to their roles in the process of developing videogames over that period. Even though this marks the sixth annual FLAs, there will be no resting for the seventh, and we're already starting to note the likely nominations for next year.

The 10 ultimate winners from that pool survived hands-on testing by our judges and earned consistently high marks on the ballots, which had separate scores for innovation, interface, ease of use, cost, and utility/integration.

In addition to these winners, Adobe Photoshop garners a long-overdue recognition as this year's Hall of Fame recipient. The Hall of Fame Award honors a product at least five years old which has proved itself indispensable to the craft of game development.

We owe immense kudos to our panel of judges, without whom we could not have put these awards together at all. They are:

Programming: Ralph Barbagallo (Flarb Development), Chris Corry (LucasArts), Jamie Fristrom (Treyarch), Shawn Green (Gearbox Software), Miguel Goncalves (Electronic Arts), Spencer Lindsay (Rockstar San Diego), Justin Lloyd (independent), Dani Sanchez-Crespo (Novarama), and Andi Smithers (Pipedreams Interactive).

Art: Tom Carroll (Rockstar San Diego), Mike Crossmire (Mythic Entertainment), Michael Dean (Ion Storm), Miguel Goncalves, and Sean Wagstaff (independent).

Audio: Aaron Marks (On Your Mark Music), Chuck Carr (Sony Computer Entertainment America; recused from judging Scream and Xact), Gene Porfido (Smilin' Pig Productions), Tom Hays (Treyarch), Todd Fay (G.A.N.G. and Tommy Tallarico Studios; recused from judging The SFX Kit), and Tommy Tallarico (Tommy Tallarico Studios; recused from judging The SFX Kit).

Game Components: Eric Dybsand (Glacier Edge Technology), Clinton Keith (Sammy Studios), Justin Lloyd, James Loe (Gas Powered Games), Albert Mack (Totally Games), William Mitchell (Imperium Games), Cary Mednick (Midway Games), and Jez Sherlock (Vicarious Visions).

Hardware and Hall of Fame: These awards were balloted by judges across all the categories.

— *Peter Sheerin*

Adobe
www.adobe.com

PHOTOSHOP

Photoshop has long been the flagship 2D texture creator and editor for game developers everywhere. While there are alternative image-editing packages, none can offer the ease-of-use and power that Photoshop has maintained since the advent of user-friendly graphics applications.

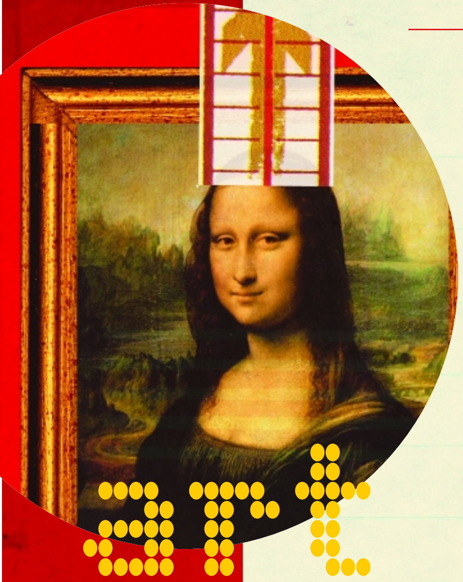
Though Photoshop doesn't possess the interface customization capabilities of some of its competitors, the interface has been evolving since its initial release, and the years of user feedback and ultimate familiarity with that interface make it simple to ignore some of the small interface hurdles.

Photoshop's greatest strength lies in the fact that there is no one right way to do a task. The depth of the package is so complete that achieving an objective can be accomplished equally well using a variety of methods. These different methods allow for artists of different mind-sets to pick and choose the path that works the way their mind works best.

Another of Photoshop's strengths is the layering system. Image layers are handled better in Photoshop than they are in any other image editor. They are intuitive and don't require abstract thought to use them effectively (for example, every brush and every tool responds the same in a layer as it does on a background).

The coup de grâce is the fact that Photoshop is widely considered the industry-standard image-editing software; therefore a huge number of companies create plug-ins to handle almost any task that lends itself to automation. Though they can be expensive, many are free, and a huge number of them are quite useful. If there's a 2D imaging task that an artist needs to pound out quickly, Photoshop is always the first package that comes to mind.

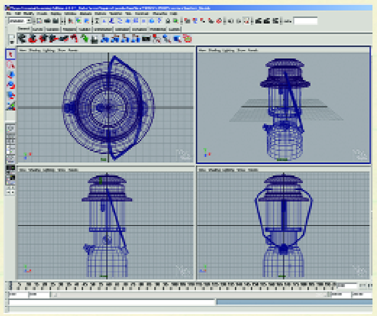
— Michael Dean



MAYA 5.0

Maya 5.0 is a powerhouse 3D application with breadth and depth to cover any 3D project's needs. The newly integrated Mental Ray adds outstanding photorealistic rendering capabilities, while new and improved polygon and subdivision surface tools make game modeling much faster and easier. Still, Maya's real ace in the hole is its amazing MEL scripting language, which lets you completely customize the program and quickly write tools to provide just about any functionality you can dream up. Maya's fast, customizable, artist-friendly interface is also an important reason to choose this program, and its new lower cost doesn't hurt either.

— Sean Wagstaff

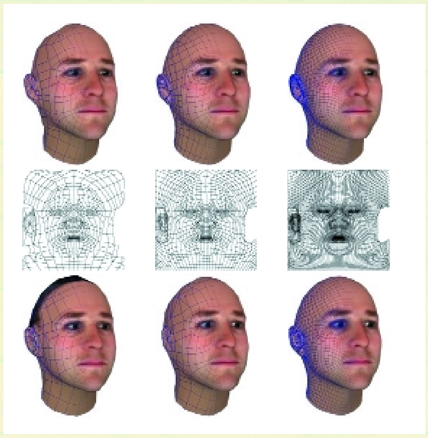


Alias
www.alias.com

FACEGEN MODELLER 3.0

FaceGen Modeller was a great surprise. The software does one thing, but it does it very well and very cheaply. Being able to generate reasonably accurate head models from photos of real-life subjects for just a few dollars a pop is something a lot of modelers have dreamed of for a long time. The mesh also comes across with a clean surface topology and UV coordinates in-place and ready to go. The resulting model is very easily exported to your favorite 3D package, and, because of the clean surface topology of the model, it is a simple matter to bring the LOD of the model down without losing many important details. Combine this with a tiny learning curve and the ability to do almost limitless variations on a head model in-software, and the package holds high value.

— Sean Wagstaff



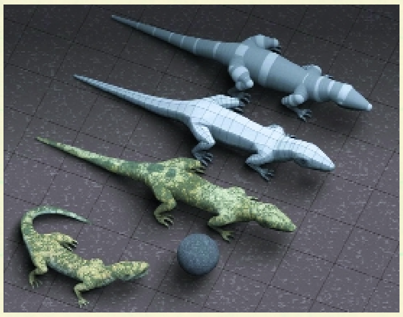
Singular Inversions
www.facegen.com

ZBRUSH 1.5

ZBrush was another wonderful surprise. I was taken aback by the depth and robustness of this package. I was expecting a simple utility but was rewarded instead with a program that nearly has enough features to compete with the big boys. I feel absolutely confident that I could do many organic models with it easier than with Max or Maya.

In addition to its rich feature-set and standards compatibility, it is also a low-profile application that doesn't tax your system for resources needed by other programs. With a bit more tuning, ZBrush can one day compete head-to-head with the industry-leading content-creation tools.

— Michael Dean

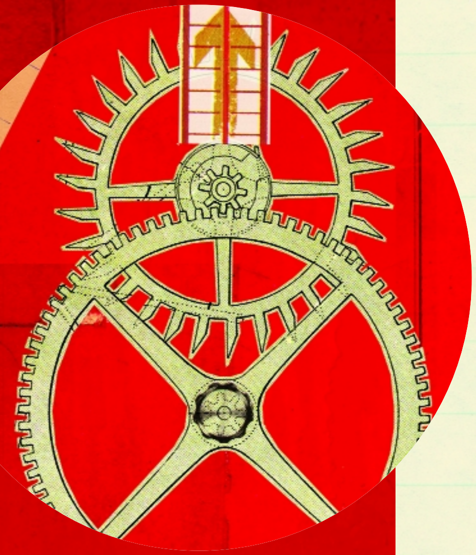


Pixologic
www.zbrush.com

FINALISTS

Softimage|XSI 3.5 — Softimage; Motionbuilder 4 — Kaydara; Kaldera for 3DS Max — Mankua; Brazil Rendering System 1.2 — Splutterfish; Npower Booleans for 3DS Max — Npower Software; Voiceworks 3.1 for Maya 5.0 — Puppetworks; Speedtree RT 1.5 — Interactive Data Visualization

game components



HAVOK 2

We know that the games we develop are becoming more complex in the game world and code base, demanding more of our available resources and time. Creating an in-house, robust physics solution can consume a lot of development bandwidth, leaving other parts of the game to suffer. With the Havok Physics SDK it's possible for game developers to conquer the next frontier being explored, physical simulation and reaction, to feed the insatiable craving of game players everywhere. Havok 2 is the most powerful, resilient, and flexible physics middleware solution available to today's game developers.

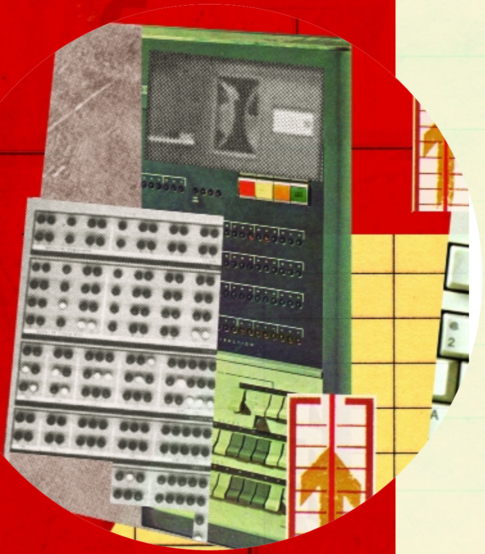
— Justin Lloyd

Havok
www.havok.com

FINALISTS

Pixomatic — RAD Game Tools; Renderware Graphics 3.5 — Criterion Software; Renderware AI 1.0 — Criterion Software; Butterfly Grid 1.6 — Butterfly.net; CRI Sofdec — CRI Middleware

hardware



VARIAX

The Line 6 Variax is a highly innovative musical instrument, modeling the sound characteristics of several popular and expensive acoustic and electric guitars, sitar, and banjo — all with incredible believability. Game composers strive to create an identity for themselves and demand the highest-quality instruments and sounds they can afford. The Variax accomplishes both demands by putting a large variety of fresh sounds right at their fingertips. This product is easy to use and can instantaneously be integrated into a game studio environment. It saves precious space and hard-earned money and is an incredible time saver, which ultimately inspires creativity rather than gets in the way.

— Aaron Marks

Line 6
www.line6.com

FINALISTS

Spacetraveler — 3Dconnexion; 01V96 — Yamaha; Noah EX — Creamware; Voiceworks — TC Electronic; Area-51m — Alienware

OPENGL ES

Right now OpenGL ES is a piece of paper. Soon, it will become a reality on mobile devices of all types. Previously, 3D on mobile devices (including phones) meant using different proprietary 3D APIs and engines. In some cases, these APIs were exclusive to the vendor, making the owner of the API the sole provider of 3D content for it. Those days are over with OpenGL ES, an OpenGL implementation for small devices. The OpenGL ES specification contains all the basic needs for 3D game developers and is surprisingly flexible. Now it's up to hardware OEMs to implement it correctly.

— *Ralph Barbagallo*

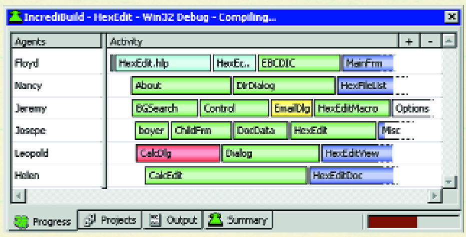
OpenGL|ES™

Khronos Group
www.khronos.org

INCREDIBUILD 1.3

Xoreax IncrediBuild is the first developer application that makes you wish your project were larger so that IncrediBuild can work its magic even better. With the ability to compile very large projects, projects larger than most people have ever worked on, not in tens of minutes, not even in a few minutes, but in just one or two minutes, this package really does save time and more importantly money. At the end of the project, when compile times using the traditional build method have slowed your progress to a crawl, you'll be wondering how you ever lived without this package in your development toolbox.

— *Justin Lloyd*

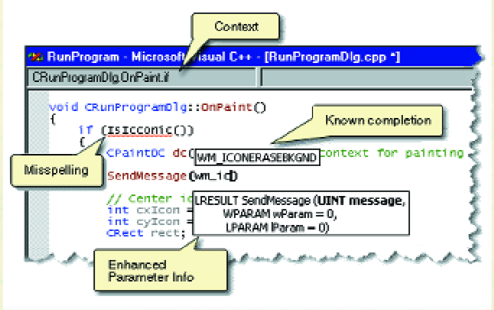


Xoreax Software
www.xoreax.com

VISUAL ASSIST .NET 7.1

A sign of a good tool is one that if you disable it, you can't continue without it. Visual Assist falls distinctly into this category. Within five minutes of disabling this very useful GUI enhancement tool for Microsoft's Visual Studio .NET compiler, you're aching for those helpful little features, like full color syntax. Amazingly, although Visual Studio is in its seventh incarnation, it still can be radically improved by integrating Whole Tomato's superb utility. Auto-corrections are almost effortlessly modified, while class browsers and context searching help navigate rapidly around complex code bases. It's also extremely fast and doesn't bog down the compiler or the IDE with annoying refreshes. All in all, a very tight little tool.

— *Andi Smithers*


Context
RunProgram - Microsoft Visual C++ - [RunProgramDlg.cpp *]
CRunProgramDlg.OnPaint.if
Known completion
Misspelling
Enhanced Parameter Info

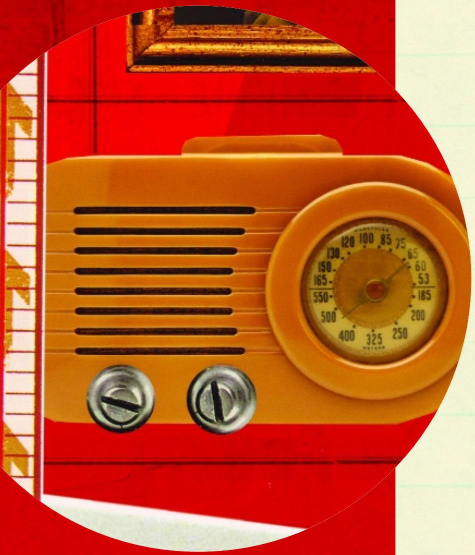
```
void CRunProgramDlg::OnPaint()
{
    if (IsIcconic())
    {
        CPaintDC dc(WM_ICONERASEBKGND context for painting

        SendMessage(wm_id)

        // Center icLRESULT SendMessage (UINT message,
        int cxIcon =        WPARAM wParam = 0,
        int cyIcon =        LPARAM lParam = 0)
        CRect rect;
```

Whole Tomato
www.wholetomato.com

FINALISTS

Visual Studio .NET 2003 — Microsoft; Perforce 2003.1 — Perforce Software; Alienbrain 6.0 — NXN; DirectX 9.0 & HLSL — Microsoft; BREW 2.1 — Qualcomm; MIDP 2.0 — Sun Microsystems; Developer's Suite for J2ME 2.0 — Nokia

audio



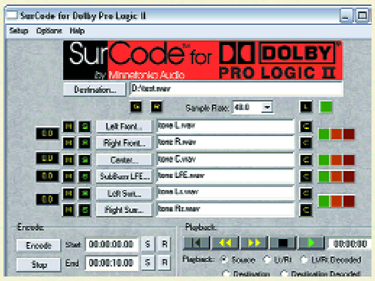
THE SFX KIT

Sound Ideas/
Tommy Tallorico Studios
www.sound-ideas.com

The SFX Kit is without a doubt the most useful audio product for the game community released this year. Not only are the more than 20,000 samples stored in industry-standard WAV format, but practically every professional-grade sound effect can be used off-the-shelf, plugged straight into a game. Whether these highly creative sounds are used stock or as elements to make fresh, innovative sounds, this library has everything discriminating game sound designers or developers would need to get the job done. This product gets an enthusiastic recommendation to anyone in the video-game business serious about standing out from the rest of the crowd.

— *Aaron Marks*

SURCODE FOR DOLBY PRO LOGIC II



Minnetonka Audio Software
www.minnetonkaaudio.com

Take a complex and expensive hardware encoding unit, turn it into a simple and inexpensive piece of software, and you have a winner. Surcode for Dolby Pro Logic II will receive nonstop use in any game studio. It is simple to use, does the job flawlessly using existing equipment, and takes what can generally be an awkward task and really makes it an invisible part of the process. In contrast to today's mindset of "the more complex, the better," Surcode's refreshing simplicity makes it a pleasure to use.

— *Aaron Marks*

FINALISTS

Nuendo 2.0 — Steinberg; VSL Complete Orchestral Package, Pro Edition — ILIO Entertainments; Scream 1.0 — Sony; Xact — Microsoft; Peak 4.0 — Bias; Digital Performer 4.1 — MOTU; Urban Atmospheres — Steinberg; Absynth 2.0 — Native Instruments

Return

Let There Be Clouds!

Fast, Realistic Cloud-Rendering in MICROSOFT FLIGHT SIMULATOR 2004: A CENTURY OF FLIGHT

You're standing on rolling hills beneath a brilliant blue sky. You look up and see huge spherical white blobs suspended a few thousand feet in the air. What's wrong with this picture? Perhaps you could use a better cloud-rendering system.

In videogames that simulate outdoor reality, realistic clouds can be one of the most compelling aspects of the scene. Clouds can also set the mood — dark thunderheads for an ominous scene, light puffy clouds for a happy mood. Michelangelo spent years perfecting the heavens on the ceiling of the Sistine Chapel, but we need to render realistic clouds in milliseconds. Fortunately, we have more advanced tools to work with. This article describes the cloud modeling and rendering system that ships with MICROSOFT FLIGHT SIMULATOR 2004: A CENTURY OF FLIGHT.

Clouds in the real world consist of many types, such as altocumulus, stratus, and cumulonimbus, and cloud coverages ranging from a few sparse clouds to a dense, overcast sky. Our cloud system models this range of cloud types and coverages. MICROSOFT FLIGHT SIMULATOR allows users to download real-world weather and see current weather conditions reflected in the game graphics, which means we need to generate compelling visuals to match any scenario that could occur in the real world.

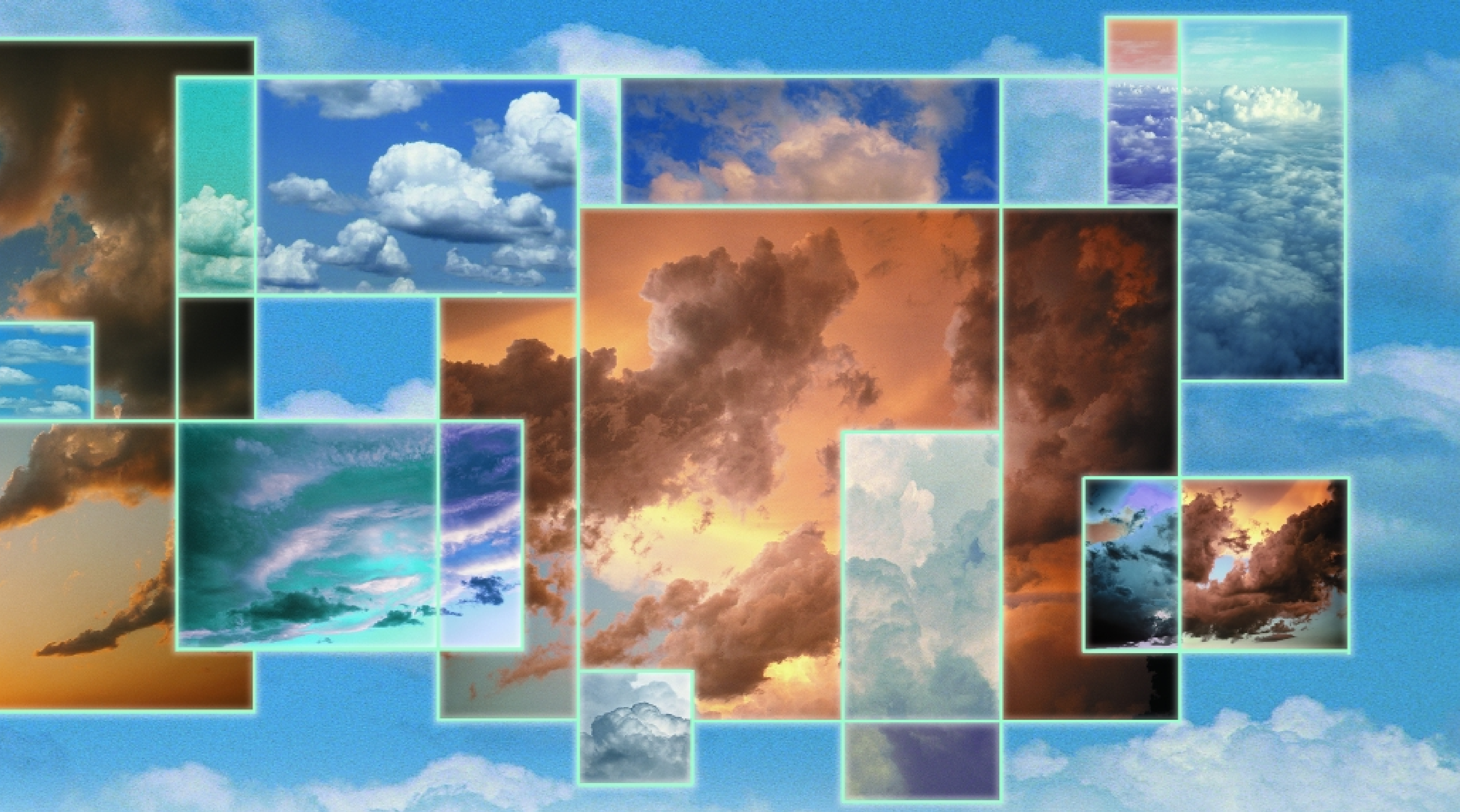
The interactive nature of games necessitates that clouds must look realistic whether the camera is far away, next to the cloud, or traveling through the cloud. Another requirement is that we need to render at high framerates. MICROSOFT FLIGHT SIMULATOR supports a wide range of machines, from the latest PCs to those dating back several years, and the performance must scale to this spectrum of machines.

Clouds need to be shaded appropriately to emulate both sunlight and light reflected from the sky, especially for games such as MICROSOFT FLIGHT SIMULATOR, which take place over the course of day, spanning dawn, midday, dusk, and night. We model the dynamic aspect of clouds by introducing a method to form and dissipate them over time.

Previous Work

Over the past 20 years, graphics researchers have modeled clouds in many ways, including cellular automata, voxels, and metaballs. They also modeled cloud animation via fluid dynamics. There are two reasons that these research techniques have not been widely adopted by games. The first is performance. Many of these systems produced screenshots that were gorgeous but required multiple seconds to render. The second is lack of artistic control. Imagine that you create a cloud by running a set of fluid dynamics equations. You examine the results and decide you would like a wispier top on the cloud. You must then iterate through cycles of adjust-

NINIANE WANG | *Niniane has five years of game development experience at Microsoft Game Studios. Most recently she was a software engineer lead on MICROSOFT FLIGHT SIMULATOR 2004: A CENTURY OF FLIGHT. She can be reached at niniane@ofb.net.*



ing variables such as air humidity and temperature, and recomputing the equations, which can require hours and still may not produce the visual effect you had in mind.

Realistic results in cloud shading have been achieved by simulating the scattering of light by particles as it passes through the cloud, known as anisotropic scattering. This produces accurate self-shadowing and interesting effects such as the halo when the cloud lies between the camera and the sun. We created a simple shading model for our system, forgoing these effects in exchange for fewer computations and higher artistic control.

Many flight simulation games have featured clouds, recent examples being FLIGHT SIMULATOR 2002, IL-2 STURMOVIK, and COMBAT FLIGHT SIMULATOR III. A common approach is to paint clouds onto the skybox texture, which has minimal performance overhead, but such clouds look two-dimensional and never get closer as the camera moves toward them. A better solution is to draw each cloud as a single facing sprite. This solution looks realistic from a stationary camera but produces anomalies as the camera rotates around it. A few recent games use clusters of textured particles, similar to our system. Some use unique textures for every cloud, which has a high video memory cost as the number of clouds in the scene increases. Other systems use small blurry textures, which results in clouds that look volumetric but lack definition. All of these systems also lack the ability to form and dissipate clouds.

Our system was inspired after hearing a GDC talk by Mark Harris, who developed Skyworks, a real-time system that created volumetric clouds from sprites. Harris dynamically generated an impostor for every cloud and achieved speeds of 1 to 500 frames per second. He also modeled the fluid motion behind cloud animation. The limitation of his system is that it cannot render large clouds, such as cumulonimbus, or dense scenes of overcast clouds, due to the prohibitively high video memory cost of generating large impostors. Our system is able to address this limitation. In addition, we tackle the problem of scaling to multiple cloud types.

Cloud Modeling

Given that we want immediate visual feedback and full control over the final result, how can we design the artistic pipeline for modeling clouds? We model each cloud as five to 400 alpha-blended textured sprites. The sprites face the camera during rendering and together comprise a three-dimensional volume. We render them back-to-front based on distance to the camera.

We wrote a plug-in for 3DS Max that creates cloud sprites based on a 3D model composed of boxes. The artist denotes a cloud shape by creating and placing a series of boxes, using default 3DS Max functionality. The artist can create any number of boxes of any size and can choose to overlap the boxes.

The plug-in UI contains an edit field to specify the number of sprites to generate. To create denser clouds, the artist would set a number which is proportionately higher than the size of boxes in the model. Wispier clouds would be created by setting a lower number. There are generally 20 to 200 boxes for each 16-square-kilometer section of clouds, and the number of sprites per box can vary between 1 to 100, depending on the density. The UI also allows the artist to specify a range for the width and height of each sprite, and choose between categories (such as stratus and solid cumulus) that determine the textures that will be automatically placed by the tool onto the sprites. Figure 1 shows a screenshot of the tool UI.

The artist presses a button in the plug-in UI to generate the cloud sprites. The plug-in creates a list of randomly placed sprite centers, then traverses the list and eliminates any sprite whose 3D distance to another sprite is less than a threshold value (the "cull distance"). This process reduces overdraw in the final rendering and also eliminates redundant sprites created from overlapping boxes. We have found that a cull radius of 1/3 of the sprite height works well for typical clouds, and 1/5 to 1/6 of the sprite height yields dense clouds. Figure 2 shows screenshots of a cloud model made of boxes and its corresponding sprites.

The plug-in creates an initial model of sprites, and the artist can now edit them within 3DS Max. Having achieved the desired visual look, the artist uses a custom-written exporter to create a binary file containing the sprite center locations, rotations, width, and height, along with texture and shading information. These files are loaded during game execution and rendered.

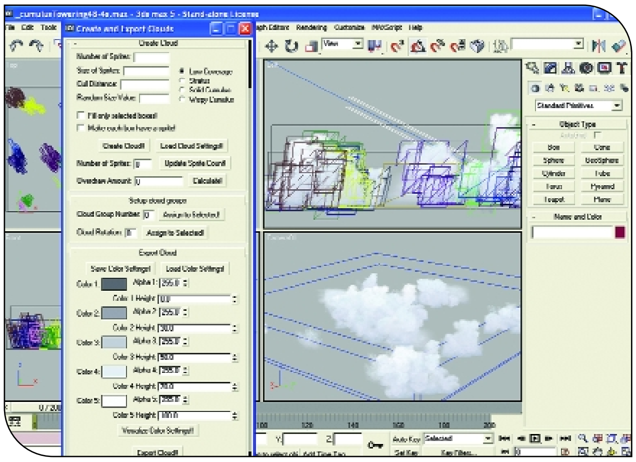


FIGURE 1. A custom tool within 3DS Max allows the artist to set properties and generate cloud sprites.

Textures

To create a dozen distinct cloud types, we mix and match 16 32-bit textures for both color and alpha (see Figure 3). The flat-bottomed texture in the upper right-hand corner is used to create flat bottoms in cumulus clouds. The three foggy textures in the top row are used heavily in stratus clouds and have a subtle bluish-gray tinge. The six puffy textures in the bottom two rows give interesting nuances to cumulus clouds, and the remaining six are wispy sprites that are used across all cloud types.

By creating interesting features inside the textures that resemble eddies and wisps, we are able to create more realistic looking clouds with fewer sprites. We place all 16 textures on a single 512×512 texture sheet, which spared the cost of switching textures between drawing calls to the video card. We automatically generate mip-map levels for this texture from 512×512 down to 32×32. To create more variations from these 16 textures, the artist specifies a minimum and maximum range of rotation for each sprite. When the binary file is loaded into the game, the sprite is given a random rotation within the range.

In-Cloud Experience

We would like a seamless in-cloud experience that looks consistent with the cloud's appearance as viewed from the outside, which does not often come with the commonly used technique of playing a canned in-cloud animation. In our system, as the camera passes through a sprite, it immediately disappears from view. We encountered a problem because the sprites rotated to face the camera, and during the in-cloud experience, the camera was so close to the sprite center that

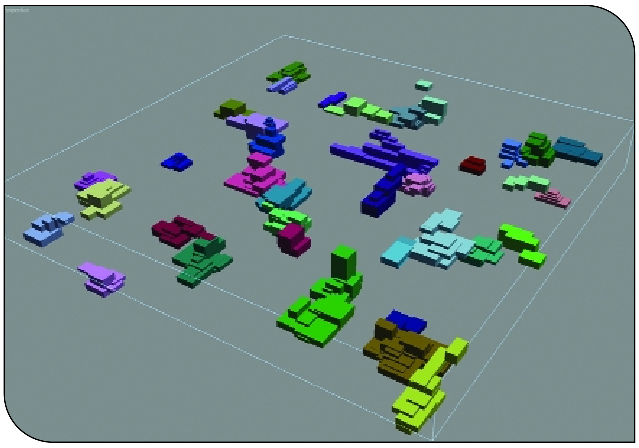
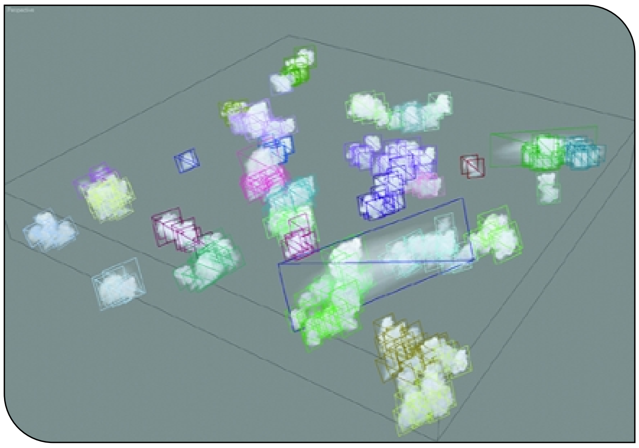



FIGURE 2. Artists use boxes to build the shapes of clouds (left), and then let a custom tool in 3DS Max populate the boxes with sprites (right).

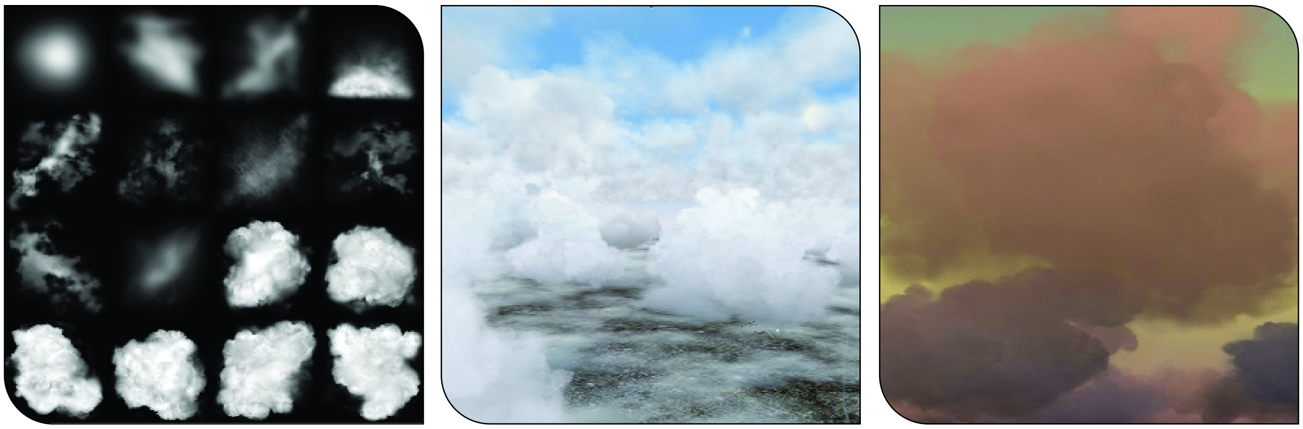


FIGURE 3. The 16 textures used to create a dozen cloud types (left) and two cloud varieties: stratus cumulus (middle) and cumulus congest (right).

small movements in the camera position caused large rotations of the cloud sprite. This resulted in a "parting of the Red Sea" effect as sprites moved out of the way of the oncoming camera.

We locked the facing angle of the sprite when the camera came within half of the sprite radius. This removed the Red Sea effect but caused sprites to be seen edge-on if they locked and the camera then pivoted around them. Our solution was to detect the angle between the sprite's locked orientation and the vector to the camera, and to adjust the transparency of the sprite. The negative side effect is that the section of the cloud near the camera appears less opaque.

Performance

MICROSOFT FLIGHT SIMULATOR 2004 maintains framerates of 15 to 60 frames per second on consumer PCs, even in overcast scenes. We achieve this across a range of machines with CPU speeds from 700MHz through 3.0GHz. Predictably, the machines with older CPUs and video cards pose a particular challenge.

The heavy amount of overdraw in the clouds presents an opportunity to improve performance. We use the impostor technique invented by Gernaut Schaufler (see For More Information) of dynamically rendering multiple clouds into a texture that we then display as a billboard. We create an octagonal ring of impostor textures around the camera, each with a 45-degree field of view. We can render hundreds of clouds into a single impostor. Our system compares clouds in 16-square-kilometer blocks against the ring radius and renders only the sections beyond the radius into impostors. Cloud blocks within the radius are rendered as individual sprites (see Figure 4).

We allow the user to set the ring radius. A smaller ring gives better performance but more visual anomalies, which I'll discuss later. A larger ring means fewer anomalies but less performance gain from the impostors, since fewer clouds are rendered into them.

We render the eight impostors in fixed world positions facing the center of the ring, and re-create them when the camera or sun position has changed past threshold values. We recalculate all eight rather than a lazy recomputation, in case the user suddenly changes the camera orientation. Empirical results show that the user can move through 15 percent of the impostor ring radius horizontally or 2 percent of the ring radius vertically before recalculation becomes necessary.

To prevent variability in framerate when rendering to impostors, we spread out the impostor calculation over multiple frames. For video cards that support it, we do a hardware render-to-texture into a 32-bit texture with alpha over eight frames, one for each impostor. For the other video cards, we use a software rasterizer to render into the texture over dozens of frames, one 16-square-kilometer cloud block per frame. When we update to a new set of impostors, we cross-fade between the two sets.

We translate the impostor texture vertically up or down based on the angle between the clouds and the camera. When the clouds are displaced more than 10,000 feet vertically from the camera, we stop rendering them into impostors because the view angle is too sharp. The overdraw is much less when

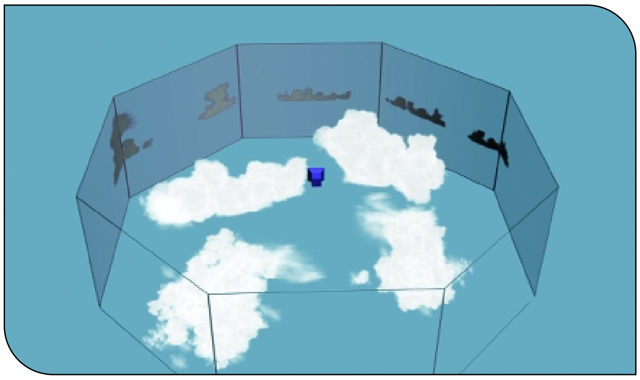


FIGURE 4. An octagonal ring of impostors around the camera eyepoint helps improve performance.

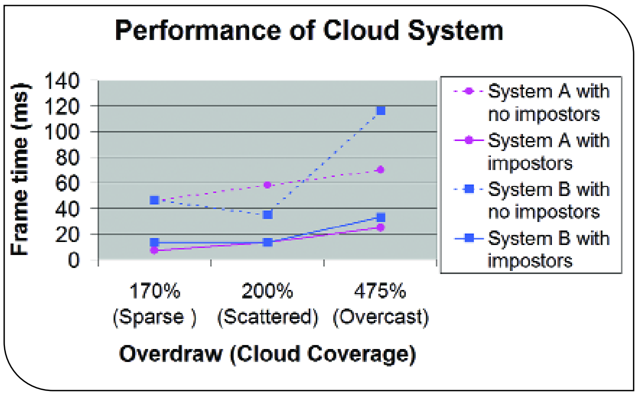


FIGURE 5. Performance comparison, with and without imposters, between System A (1.7GHz Intel Pentium, 768MB RAM, GeForce2 GTS) and System B (733MHz Intel Pentium III, 128MB RAM, Riva TNT2).

the clouds are far away, so performance in this situation only suffers slightly by not rendering into impostors.

Since video memory is frequently a tight resource on consumer PCs, we designed the impostor system to have low video memory usage. Our ring of eight impostors, each a 256×256 texture with 32-bit color, adds up to a video memory cost of 8 × 256 × 256 × 4 = 2 megabytes. When crossfading, both impostor rings are rendered, which adds another 2 megabytes during the transition.

Figure 5 shows our framerate for three scenes of varying coverage, on two machines. We chose older machines for this experiment to show that our results scale to machines with slower CPUs and lower video memory.

System A is a 1.7GHz Intel Pentium with 768MB of RAM and a GeForce2 GTS video card. System B is a 733MHz Intel Pentium III with 128MB of RAM and a Riva TNT2 video card.

On older systems, such as 450MHz machines with 8MB video cards, the fill rate is so expensive that even with the impostor ring radius at eight kilometers, rendering the cell block within the ring radius as individual sprites produces framerates that fall below 20 frames per second for denser cloud coverage. We created a simple LOD scheme that uses a single sprite per cloud. Since our rendering method scales to any number of sprites in the model, a single-billboard cloud is merely a degenerate case of the model, and we needed no special-case code to render or shade these models.

Shading: Every Cloud Has a Silver Lining

We chose not to simulate scattering of light from cloud particles, instead using simple calculations based on artist settings that yield a reasonable approximation. This means we do not simulate clouds casting shadows on themselves, other clouds, or other objects in the scene. The two factors that go into our cloud-shading system are skylight and sunlight.

As rays of light pass from the sky through the cloud, they are scattered and filtered by the particles within the cloud. As a result, clouds typically have darker bottoms. To simulate this, our artists use a color picker in 3DS Max to specify five "color levels" for each cloud. The color level consists of a height on the cloud with an associated RGBA color. These levels are exported in the cloud description file.

Separately, for a set of times throughout the day, the artist will specify a percentage value to be multiplied into the ambient color levels at each particular time of day. This allows the ambient contribution to decrease approaching night.

The sun casts directional light on a cloud, which generates dramatic scenes, especially around dawn and dusk. We simulate the effect so that areas of the cloud facing the sun receive more directional light while areas facing away from the cloud receive less.

Our artists specify shading groups, sections of one of 30 sprites that are shaded as a unit, when they build the clouds in 3DS Max from boxes. On each box, they set a custom user property with a shading group number, and sprites generated for that box will belong to that shading group. These shading groups simulate clumps on the cloud. We calculate the directional component of shading for a given vertex in the cloud by first computing the vector to that point from the shading group center. We also find the vector from the group center to the sun and compute the dot product of the two vectors after normalization.

The artist specifies a maximum directional color and minimum and maximum directional percentages for a range of times throughout the day. We multiply the resulting percentage from the mapping function with the maximum directional color specified by the artist, to get the directional color of the vertex.

To get the final vertex color, we add the ambient and directional colors to the color from the sprite texture. At this point, we also multiply by the alpha value representing formation or dissipation of the cloud.

Formation and Dissipation

The clouds look more realistic when they can form and dissipate. We control the evolution of a cloud by adjusting the transparency level of sprites.

We multiply a transparency factor into the alpha value for each sprite vertex based on its position within the cloud. When a cloud is beginning to form, we render only the sprites whose center is within half of the cloud radius from the cloud center, and we render them with a high transparency level that we decrease over time. After these sprites have reached a threshold opacity, we begin to render sprites whose center is more than half of the cloud radius from the cloud center. Cloud dissipation is simulated by reversing the process (see Listing 1).

LISTING 1. Calculating the alpha value of a vertex when forming and dissipating the cloud.

`vertex` is the (x, y, z) with respect to cloud center.
`cloud_radius` is the radius of the cloud bounding box.
`alpha_at_edges` controls how much to fade out the edges; this increases from 0 to 255 over time.
`alpha_of_cloud` controls the transparency of the entire cloud; this starts out at 255.
`time_delta` is the time that passed since the last frame.

```
float alpha;
if (fade_out_to_edges)
{
  float radial_magnitude = cloud_radius -
    vector_magnitude (vertex);

  radial_magnitude = max (0, radial_magnitude);

  alpha = (alpha_of_cloud - radial_magnitude * alpha_at_edges /
    cloud_radius) / 255.0f;
}

alpha = min(1.0f, max(0.0f, alpha));

if (alpha_at_edges < 255)
  alpha_at_edges += 255 * time_delta / time_to_fade_cloud_edges;
else
  alpha_of_cloud -= 255 * time_delta / time_to_fade_cloud_core;
```

Limitations and Extensions

Our system is well suited for creating voluminous clouds but less suited for flat clouds. Of the four basic cloud types — cumulus, stratus, cumulonimbus, and cirrus — our system easily handles the first three. However, cirrus clouds are so flat as to be almost two-dimensional, and it would require a large number of small sprites to model them using our approach, which would cause a performance hit. Instead, we represent cirrus clouds with flat textured rectangles.

Because sprites within each cloud are sorted back-to-front to the camera, moving the camera can occasionally result in popping as sprites switch positions in the draw order. This effect is more noticeable at dawn and dusk, when directional shading plays a greater role, but has not been jarring enough in our experience to necessitate a solution such as caching the previous sort order and crossfading.

As mentioned previously, our shading model does not include cloud shadows, self-shadowing, or the halo effect when the cloud is between the sun and the camera. One potential solution is to precalculate the lit and shadowed regions of the cloud for a set of sun angles. We can load this information at run time and interpolate based on sun angle.

The ring of impostors can create visual anomalies. The clouds in the impostor do not move relative to each other, which means the parallax looks wrong. Also, distant objects must be drawn either in front of or behind all the clouds in the impostor, instead of in front of some and behind others. We could mitigate this by adding additional rings of impostors, but that increases video memory usage.

In the future, we would like to implement some of our techniques into vertex shaders, as more of our user base upgrades to video cards that support hardware vertex shaders. Our system can be extended to other gaseous phenomena, such as fog, smoke, and fire. Fog is a natural candidate, since it is essentially a stratus layer placed close to the ground. The problem is getting rid of the hard lines where sprites intersect the ground polygons. We can either split the sprites along the ground or multiply by a one-dimensional alpha texture based on the altitude.

Research into fluid simulation has produced realistic animations of clouds as they change shape. This creates more extensive morphing than our system of formation and dissipation, but it frequently does not yield enough control to the artists over the final result. It can be difficult to tweak the humidity and other parameters just right to have a cloud form over three minutes, or to ensure the cloud that forms looks a particular way.

A solution more appropriate for games and movies may be one that combines our artistic modeling with fluid simulation by using simple rules for cloud morphing in combination with our system of textured particles. For example, we could use weather variables such as humidity, airflow, and temperature to rotate, translate, and adjust transparency on individual sprites within the cloud to change the overall shape of the cloud. It would give the impression that wisps are being blown by the wind, or that clouds are condensing or breaking into several pieces.

FOR MORE INFORMATION

Download a video describing the cloud-rendering system at www.gdmag.com.

Mark Harris and Anselmo Lastra. "Real-Time Cloud Rendering." Computer Graphics Forum, vol. 20, issue 3. Blackwell, 2001.

Gernaut Schaufler. "Dynamically Generated Imposters." Modeling Virtual Worlds — Distributed Graphics, ed. D. W. Fellner, MVD Workshop, 1995.

ACKNOWLEDGEMENTS

Thanks to Adrian Woods and John Smith, two very talented artists from Microsoft, without whom this work would not have been possible.

Return

JAK II START TO FINISH

2 years	
335,000 person-hours	
227 pages of script	
22 audio recording sessions	
12.2 pencil-miles of drawings	
841,000 lines of run-time code	
12,000 cans of Diet Coke	
1 ocean of Goldfish crackers	

JAK II PRODUCTION GOALS

- Surprise the market by taking the game in an unexpected direction.
- Age up the universe while still retaining elements of the style.
- Give Jak a voice.
- Evolve the integration of story and touch on more mature themes.
- Eliminate boring collection as a primary task.
- Leverage the mechanics that worked in the first game and add to them.

Naughty Dog's Jak II

When Naughty Dog released Crash Bandicoot for the Playstation in September 1996, the novelty of 3D gaming was still a fresh and invigorating force. The Crash series took advantage of the new 3D environments while still retaining many of the linear motifs of classic 2D design. The series was a great success, but as the transition to Playstation 2 began in earnest, Naughty Dog began experimenting with no-load, open environments, and various forms of nonlinear gameplay, which culminated in the release of Jak and Daxter: The Precursor Legacy for Christmas 2001 (see "Postmortem: Naughty Dog's Jak and Daxter," April 2002). This new game was greeted with great enthusiasm and quickly reached Greatest Hits status, leaving no doubt that we would develop a sequel. We also knew we had significant room for improvement, especially in light of the changing competitive landscape.

In the early months of 2002, the entire company met for pre-production meetings at the ski lodges of Mammoth, Calif. In between mountain board runs and bone-jarring wipeouts, we slowly devised a plan. Never content with playing the same old sequels ourselves, we didn't just want to create another game with new levels and a big fat "2" superimposed over a catchy subtitle. We decided to shock everyone with a bold change in direction.

We looked deeply at the current state of platform gaming, and although we had tremendous respect for those who had shaped the genre, we felt that after breaking through the 3D dam and flooding down the ravine of possibilities, platform games had run their current course. We saw the genre suffering from a general malaise and a waning audience, and we were convinced that this atrophy was due primarily to lack of innovation in a changing market, especially in light of new gaming paradigms that had evolved. Gone were the days when coin collect-a-thons in neon bright worlds were enough to excite players. The maturing videogame audience wanted more realistic themes and intense experiences, and platform games had the decidedly uncool stigma of G-movie kiddie fare. Naughty Dog decided that if we could make Jak II more mature, add a deeper emotional layer to the action, and increase the entertainment value of the entire experience, we could reignite interest in action platforming.

Jak and Daxter's immersive no-load system had already evolved the way players move through platform games by eliminating discrete levels bound by obvious load times. We also admired the freeform, emergent mechanics of sandbox titles such as Grand Theft Auto III. Then it hit us.

GAME DATA

PUBLISHER: Sony Computer Entertainment
NUMBER OF FULL-TIME DEVELOPERS: 48
LENGTH OF DEVELOPMENT: 2 years of full production
RELEASE DATE: October 14, 2003
PLATFORM: Playstation 2
OPERATING SYSTEMS USED: Windows NT, Windows 2000, Linux
DEVELOPMENT SOFTWARE USED: Allegro, Common Lisp, Visual C++, Maya, Photoshop, X Emacs, Sound Forge, Visual SlickEdit, tcsh, Exceed, CVS

DANIEL AREY | *Daniel is the creative director at Naughty Dog. As a senior game designer for the last 14 years, he has contributed to the success of many well-known videogame franchises, including the multi-million-selling Crash Bandicoot 2, Crash: Warped, Crash Team Racing, Jak and Daxter, and most recently Jak II. Prior to joining Naughty Dog, he honed his skills with Electronic Arts, Accolade, Sega of America, and Crystal Dynamics, where he helped create such hits as Crash n' Burn, Total Eclipse, Gex 3DO, and Gex: Enter the Gecko.*

Why not take our no-load system and build a huge, living city, then let loose a platform character like Jak in it? From this idea grew the concept for JAK II.

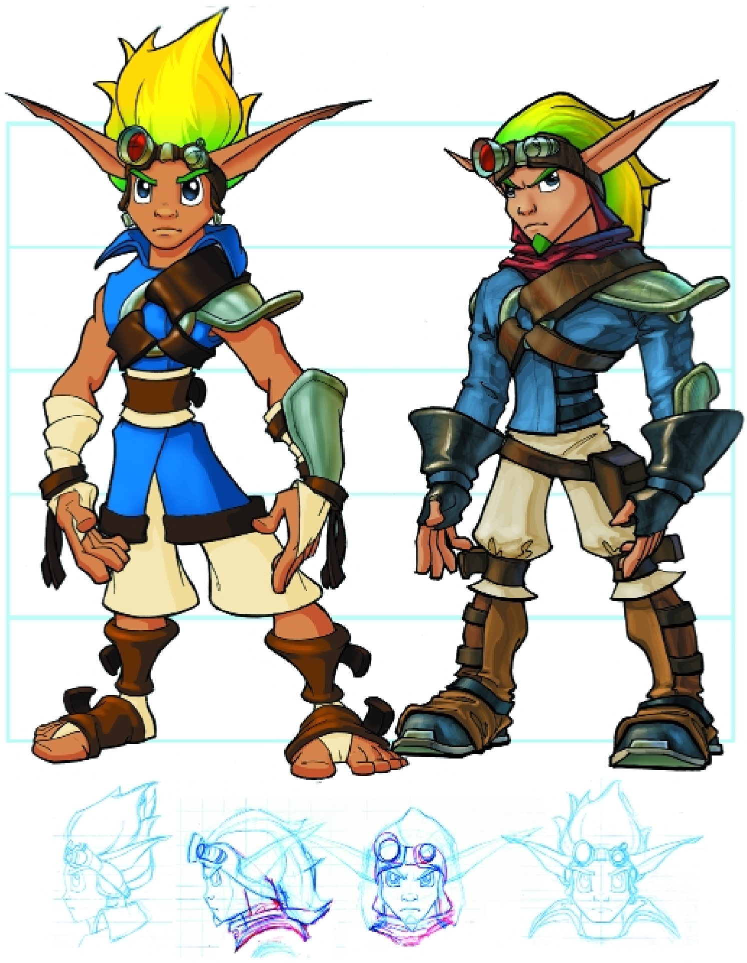
What Went Right

1. Jak gets a pair. Recognizing the changing audience, our biggest goal for JAK II was to mature the universe. We wanted to remain true to the original vision of the first game but add more maturity to the visual style. This decision required extensive exploration by the art department, starting naturally with the central hero. Since the first game was about Daxter's transformation, we decided early on that JAK II would be Jak's story and his evolution. Daxter was already a resounding success, having been named "Original Game Character of the Year" in the second annual Game Developers Choice Awards, but Jak was a different story. Extensive feedback told us that Jak had less personality than a slug on Prozac.

Jak himself was graphically redesigned to reflect a new toughness, commensurate with the grittier theme of the game. Jak's new military-inspired threads, slicked-back hairstyle, soul patch, lethal weapons, and his Dark Jak side all began to add fresh visual interest.

We also realized we'd made a serious mistake in not giving Jak a voice in the first game. Muting him had damaged his ability to interact with characters, made for awkward scenes, and left the character pixel-thin. Casting began in earnest, and Mike Erwin won the pivotal role. Jak's new voice gave him an instant personality boost, and allowed banter for the first time with Daxter (voiced by Max Cassela).

Other characters and background enhancements followed, and after many design iterations, we arrived at the new look for JAK II. The story was also given a darker edge and swam in more adult themes like revenge, love, betrayal, and death, all rare elements in platform games. Even though the game still retained much of the cartoon feel, the new designs shed some of their earlier iconic sugar coating, moving the game more toward the mainstream Playstation 2 audience.

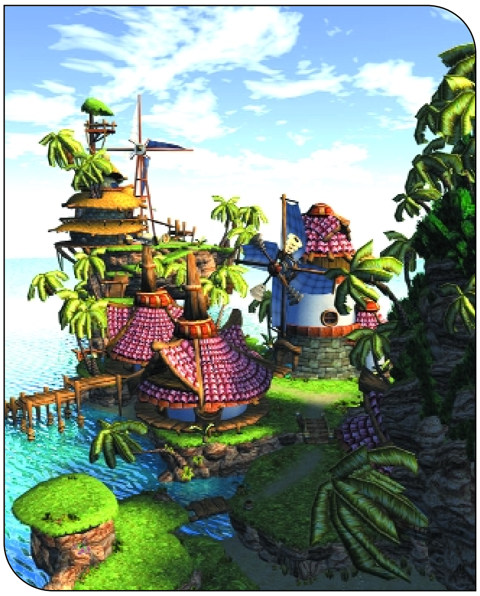
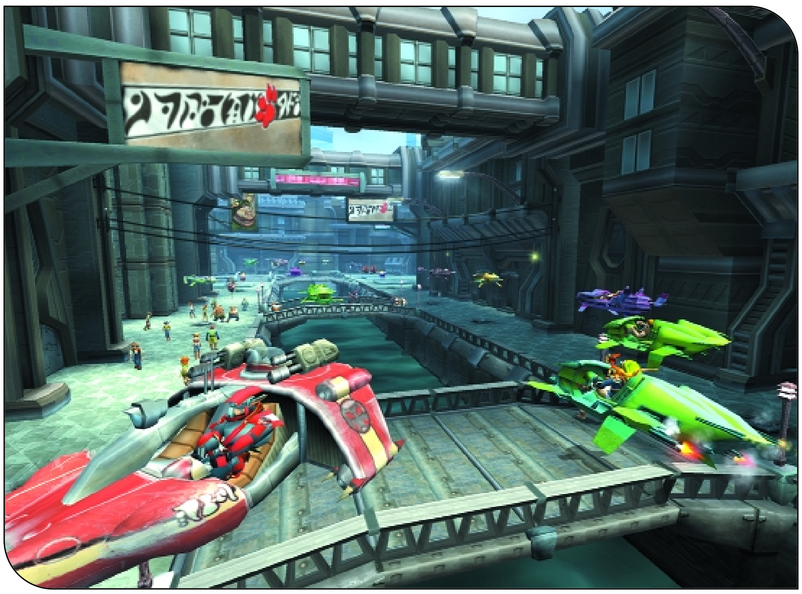

The concept art for Jak from the first (left) and second (right) games shows a subtle but clear maturing of the character.

2. Emotion in motion. Another major success in JAK II was the increased role of the story. Naughty Dog had flirted with deeper story motivation in the first JAK AND DAXTER, but in JAK II we wanted the game's story to really matter, providing true cause and effect, context, and meaning for all actions in the game. Platform game stories have traditionally been thin at best, but with the living, breathing world of JAK II it took on a whole new significance.

We wanted players to be driven forward in the game not just by the fun, but also by a sincere desire to find out what would happen next. We called this concept "narrative as reward," defining a clear system of precisely placed story plot points, special-effect cinematic sequences, and entertaining humor vignettes as the primary reward for completing each gameplay challenge.

The story evolved into 90 minutes of cinematic sequences linked inexorably to gameplay. This tight coupling between game and story required specifics in the dialogue that locked down play elements to an uncomfortable degree. As we recorded and animated the scenes, it became increasingly difficult to eliminate or adjust gameplay because of these links. To combat this, we kept the story as flexible as possible without breaking the cause-and-effect chain by having a few alternate recordings for vital plot and character revelations attached to a variety of backup gameplay tasks. We also built in what we called stand-alone levels, which, if needed, we could excise from the game (and the schedule) without greatly affecting the story experience.

In the end, the story was very well received. Players loved the ani-



The world of JAK II sheds the primary colors of the first game (left) for darker, grittier textures (right).

mated sequences and told us they looked forward to them while playing. This underscores the value of an emotional player connection in videogames, a vital aspect of future development as we move toward more sophisticated entertainment production values.

3. Fat city. Making the huge city hub for JAK II (20 times the size of a single JAK AND DAXTER level) was a massive undertaking but well worth the investment. By adding this "game within a game" to the platform experience, we hoped to create a new expression of the genre. The sense of place and purpose was designed to keep the player plugged into the world as never before. Gamers weren't just playing a succession of platform levels; they were visiting a "breathing" universe.

The city presented significant challenges, since it was populated with a multitude of citizens, an AI police force, and a rigid body physics traffic system, making for complicated combinatorial events. The traffic system was a complex web of interlocking paths and sophisticated AI to handle the multitude of disruptions on the system. The programmers had to manage traffic jams, cross patterns, and, the fun part, violent player actions upon the vehicles.

Memory and load constraints dictated how fast we could drive through the streets, and loading limits and line-of-sight issues forced the city to be more mazelike than we had originally intended. Since the entire city obviously couldn't fit into memory, load planes were painstakingly placed to hide the actual geometry swaps, and visual pop-ins were slowly tested out of existence (well, almost).

Also, since our city could be viewed from a variety of internal and external locations, it forced an incredibly complex system of multi-resolution geometry levels. Still, to look down at the city from the palace, or back at it from the mountain temple, made all the headaches worth it. People have told us that during their play experience, they found themselves stopping just to admire the view, or to watch a sunrise. To us, that's the definition of immersion.

4. Move set integration. Gameplay integration was another accomplishment in JAK II, our mantra being "If we give something to the player, we let them keep it and use it whenever they please." When Jak gets a new item, such as the JetBoard, players can (and do) pull it out wherever they want. Needless to say, this created considerable nightmares for bleary-eyed level designers trying to contain myriad combinatorial approaches to their level challenges. In the end, however, the freedom gave players a wonderful richness of play options.

Jak already had combo punches and jumps in his moveset, but in JAK II we gave him a gun and allowed him to combo that as well. Jak can jump up and shoot down, or jump up and do a helicopter-spinning rapid-fire attack. He can punch with the gun, or perform hit-and-shoot rapid-fire combos, all with fluid interpolation of animation. As a result, Jak has a total of 373 unique animations, using interpolation blending techniques to smooth the transitions.

Shooting in third person can often be suboptimal because of the imperfect limitations of visual information and control for the player. In JAK II, we wanted the mechanic to be skill-based, while simplifying the action of drawing a bead on an enemy, without intruding heavily on the fast action aspects demanded of platform challenges. To this end, we chose an auto-assist gun mechanic that reasonably solved this dilemma.

These efforts added up to choices for the player and an unprecedented level of anytime/anywhere move integration with evolving toys. Players love freedom, and they especially love it when a world lets them choose a variety of solutions for the challenges they face. The multiplicity of moves and philosophy of anywhere-use objects in JAK II allowed that freedom to flourish.

5. We got game. Ultimately the diversity of JAK II's gameplay became its biggest strength and our proudest achievement. It's difficult to label JAK II simply a platform game, since there are so many gameplay types folded together. Jumping isn't the primary player response in JAK II. You also get to drive multiple vehicles in a huge city (including a stadium with high-speed race courses), create havoc with a solid run-and-gun shooting element and multiple upgradeable gun types, trick and grind anywhere with a TONY HAWK–style JetBoard, crash-and-smash the world with a Titan Powersuit, play mini-games, and work with characters in AI-assisted escort tasks. Our goal was to achieve a diversity of gameplay in JAK II that would keep the game fresh for the player, and the challenges evolving.

It appears playground worlds are here to stay, and as long as they thrive, a bold variety of play mechanics will be a great asset to open-ended gaming. Of course, caution is advised. More is clearly not always better. We have found it's best to focus on a core set of mechanics, and only after those are playing well, then add new layers of complexity. It's also important to be willing to kill your babies, mechanics that simply aren't working, even if considerable resources have been sunk into the concept. I've heard this referred to as the "alpha apocalypse," but we suggest you do it sooner than that.

What Went Wrong

1. The Babel factor. We have always prided ourselves in working closely with our worldwide partners to include their needs in our development choices, but in the case of JAK II it became more difficult than ever. JAK II's enormous story assets quickly became an unwieldy mess as over four hours of in-game speech and cinematic sequences, broken into hundreds of VAG files, required localization for seven languages.

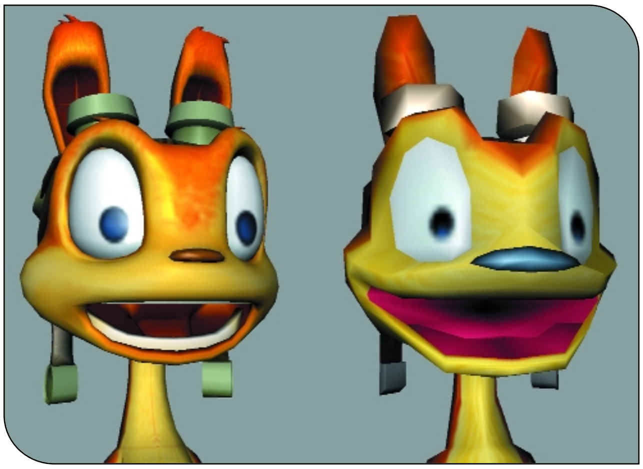
The sheer volume of files for all languages moving in and out of our office quickly became hard to track, especially when late dialogue changes were called for. To help combat this immense data tracking complexity, we created a Spoken Audio List that combined in one place every single dialogue line in the game, who said it, where it was said, and where is was in terms of U.S. recording, localization, in-game placement, filters and effects, and so on. This list grew into a huge document, but it became our invaluable audio bible, and we prayed to it daily. Without this comprehensive list in one place we would have been unable to track the vast flood of VAG files as they came and went.

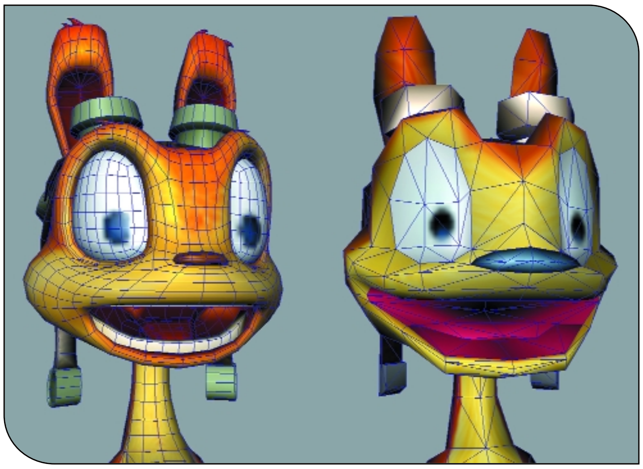
Even with the audio bible, we wasted valuable production time and gained a good number of ulcers fighting needless audio fires, and without some last minute heroics, we would have been in trouble.

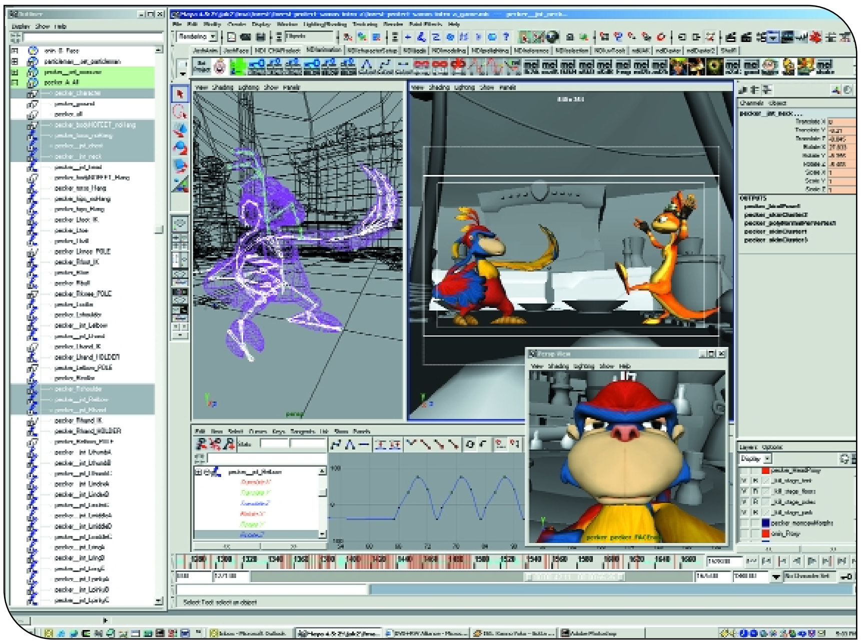
2. Facing the music (again). Our Achilles' heel of late, the sound and music spec became a monstrous data-wrangling challenge. The sheer quantity of in-game events, foley effects linked to movies, and various other sound issues became a logistical nightmare. Add to that the in-game music and scoring for the large array of cinematic scenes, and you can see how quickly the problem multiplied. The biggest sound challenge in open gaming is that you can't simply load a discrete bank of sounds in a closed environment and call it a day. JAK II's open environments were always evolving, so we had to devise a complex system of bank swaps and loads.

Naughty Dog's in-house sound group burned vampire hours to fill the world with audio cues and stuff them into tight mem-

JAK II's models, shown on the left of the images below, contain 10,000–15,000 polygons each, far more than in the first game (right images).







Up to six animators took over 14 months to complete JAK II's 90 minutes of cinematic sequences using Maya.

ory constraints. Also, since some of our movies were being finished very late, we had our foley cinematic group working around the clock to finish the multitude of effects across 90 minutes of movie. We left them too little time in the end, and we missed some very important sound cues. The simple solution here: Don't send game objects and movies for sound and foley work right up until the last few days of production, and don't expect inhuman man-hours to make up the difference.

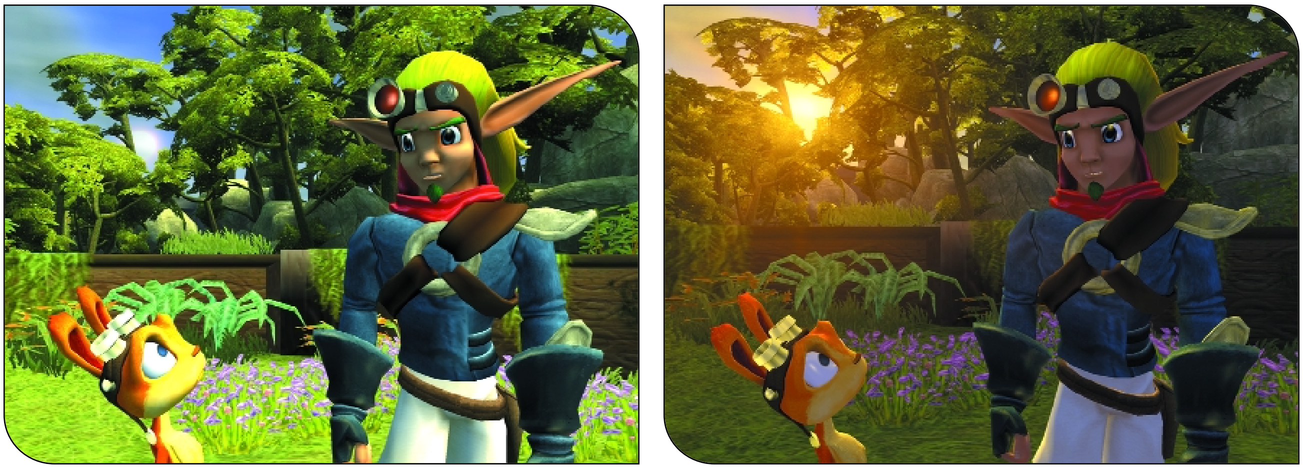
Once again our game music, although very well orchestrated, lacked enough depth to sustain the entire game. Since music is such a vital aspect of emotion and mood, and Jak's story turned in a number of directions, we failed to adequately cover the range of playtime the average player would experience. Also, due to our constant game data streaming, JAK II's in-game music was limited to MIDI, and this became a sad limitation when the scope of the adventure demanded so much more. We experimented with a range of channels with interactive elements to vary the mood depending on context (for instance adding a drum beat overlay when Jak pulls out the gun), but many people felt some of it was too heavy handed, and we ended up minimizing much of the work.

We had every one of these problems on JAK AND DAXTER, so I'm afraid we can't plead ignorance this time. Once again we underestimated the volume of work and allowed ourselves to be understaffed, only adding people to fight fires when it was much too late. Pulling up the rear is never an easy task, but our sound group was left dangling in the wind and worked diligently to mitigate bad planning on our part. This is an area of production we will move up in priority next time.

3. Read my lips. Staying ahead of the game curve was difficult for the animation department as well. Rework became a serious drag on productivity and a blight on the schedule. One example was when we lost time reworking a number of animations that spelled out specific numbered gameplay requirements in the dialogue. For instance, we created a scene where a crime boss character named Krew told Jak how many money bags he needed to pick up in a task around town. For safety, we recorded a spread of numbers, but after cutting the scene, blocking, animating, and lip-synching, our game testing revealed that we needed to adjust the number. Changing cases like this were an understandable source of frustration for the animators.

As we improved our methodology, we learned to be more generic with certain gameplay-specific numbers in the dialogue and to use screen overlays to convey details. When we really needed a specific number cited in a scene for some reason, we recorded many, many variations to give the designers and animators enough coverage. Then we blocked the shot so that the character speaking the offending (and likely variable) line could be either off-screen or conveniently in an over-the-shoulder shot, effectively hiding his lips, when the high-risk line was spoken. These techniques allowed any required changes to be easily dropped into the AIF later with no impact whatsoever on the animation side. Still, we lost considerable animation time to reworking before we defined and implemented these tricks.

4. Particle man, particle man. JAK II's particle effects grew to become extremely elaborate. They also became a nasty bottleneck at the end of the production pipeline. Special-effects programming can be one of the most cost-effective investments in terms of bang-for-the-buck polish and overall environmental look to a game. Particles add incredible life to a scene: moving smoke, dust bowls, heat effects, particulates in the air, and even insects and birds can flesh out environments with movement and verisimilitude.



By blending various directional palettes with an ambient palette and deriving colors from a mood table list, Naughty Dog could present a wide range of times of day.

Taking this lesson from JAK AND DAXTER, we jumped in with both feet on JAK II — which turned out to be the problem, as two feet meant only one person. We had some great particles very early, but as we went along, we realized that we needed more effects than we had budgeted for. Also, as the new high-polygon sets for cinematics came online, they too required a larger coverage of particle effects than we had expected. Add to that a breakdown in communication between departments about particle node placement and volume of particles in all levels, and some objects being completed very late, and you can see how the problem quickly escalated. We added a second particle programmer in the end, but when the dust settled (insert particle effect here), it was too late, and we were forced to compromise the total number of effects in the game.

5. Leveling the playfield. Bringing backgrounds to tangible life from flat 2D level maps is an art unto itself. Unfortunately, we often added to the angst of the background artists by designing area maps with too many unique elements, or worse, unrealistic spatial relationships. Memory being an ever-present constraint, the background artists had a challenging time taking a designer's level map and remaining true to the gameplay distances while still serving the production art guidelines for the level.

Naughty Dog uses an instanced geometry-building system of "ties" with carpeted "t-frag," and the artists wrestle to represent JAK II's world with as few pieces as possible to achieve critical mass and balance in terms of design precision of gameplay, memory, and aesthetic beauty.

The first design maps were too complicated, and early levels needed to be reworked, costing valuable production time. We attempted a system of rapid prototyping for levels but never adequately solved the rework problem. The design department had to learn to create more simplified maps and trust the artists to fill in the variation.

As always, art is illusion, and our background artists began to work miracles creating complexity with only a few building-block instances of repeated geometry. By taking just a few "ties" and linking them together in a wide variety of 3D patterns, they were able to achieve so much more with less.

Ultimately, we became better at designing with a delicate balance between unique and repetitive elements, but not before considerable time was lost in the learning curve, and these losses caused a chain reaction of schedule adjustments, forcing us to scale back some game areas.

Jumping Beyond Platforming

Despite the changing landscape of gaming, JAK II represents two hard years of work by a talented team of professionals dedicated to bridging a widening gap between old and new gaming. It was a tough haul, and we probably bit off more than we could chew, as many of the What Went Wrongs indicate. However, in the final analysis, we feel JAK II is the best game Naughty Dog has ever produced, and we are proud of the entertainment experience it represents.

We recognized the need to shed old assumptions about the platform genre and to try to reenergize a wonderful style of game that we all grew up with and love. Just as this year's Game Developers Conference slogan both challenges and warns us to "Evolve," so too platform games must heed the call or risk being relegated to the dusty $5 bargain bin of life. We'd prefer evolution, and we hope JAK II is a valuable step down that path.

At Naughty Dog we are now convinced more than ever that the quick reflex and button action of platform games, in whatever evolved forms they take, will always entice, entertain, and challenge those who love a great playground. At the time of this writing, JAK II is the fastest-selling release we've ever had, beating our previous sales champion, CRASH 3: WARPED, by over 50,000 units in the first two weeks. We hope this trend continues and provides exciting proof that the market for platform action games, in all their hybrid forms, is still very much alive and kicking.

Return

matthew sakey

The Hidden Language Of Genre

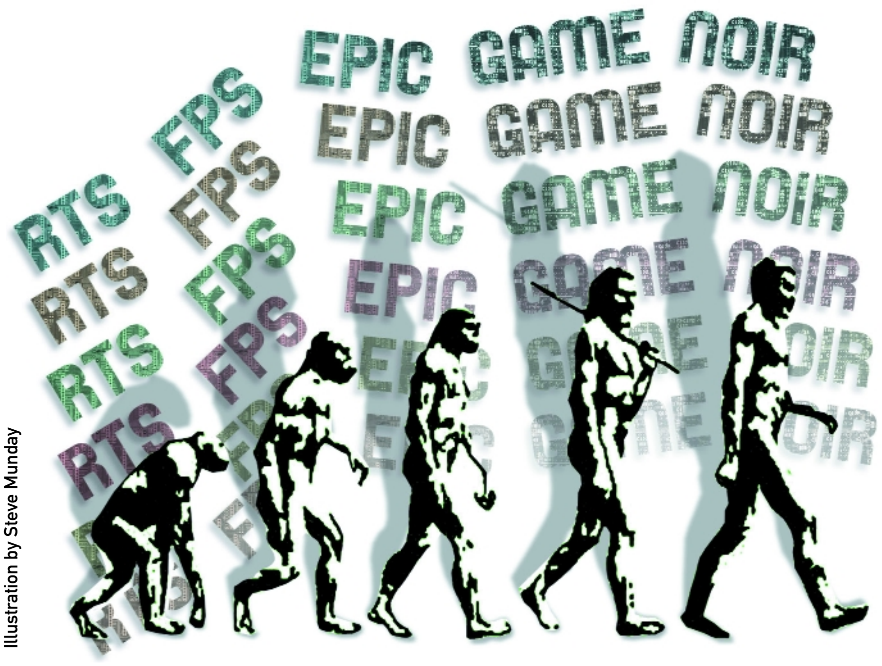


Illustration by Steve Munday

The misapplication of the term "genre," or rather the way that the game industry uses the word, limits thematic innovation and can be damaging to retail success. Shiny's SACRIFICE performed poorly at retail due in part to the fact that it was crammed into a "genre," real-time strategy, with which it shared essentially nothing. The WARCRAFT crowd found SACRIFICE bewildering, and it missed its real target. Similarly, the MAX PAYNE franchise is described as a third-person shooter "genre" game. While stylistically correct, this description shortchanges the thematic and emotional impact of the game design.

In movie terms, genre theory refers to the study and classification of films sharing stylistic, symbological, and structural components. Genre theory seeks to classify a narrative's thematic intentions through examination of the manner in which these elements are applied. In videogames, "genre" most often describes control structure, victory conditions, and presentation of the environment, when in fact these criteria are styles of gameplay: "shooter" and "puzzle" are not genres. This misapplication prevents the development of genre theory for games, a deficiency which cripples narrative evolution in the medium.

Problems of Distinction

In other media, genres are classified by creative structures. Game genres are distinguished technically. We don't describe "paperback" and "hardcover" as the genres of literature, but that's the equivalent of the current classification system for games. Technical specifications evolve independently of creativity, and to use them as genre differentiators damages the medium's creative advancement.

This is not an attempt to get stores to rearrange their shelves. Organization by style is more sensible than by theme. But the current language is inadequate for game development's creative ontology. As game narratives increase in complexity, it becomes correspondingly more challenging to develop intricate thematic experiences. Lacking mature genre theory, developers can't unleash the full narrative or interactive potential of their games, just as early filmmakers lacked such tools, depending instead on pure motion pieces or hackneyed melodrama.

Film pioneer Louis Lumiere believed film was "a medium without a future." As the generations of students who've studied Lumiere films know, the statement was valid at the time because genre theory didn't exist yet. Lumiere films examined movement, and there's a limit to how often a moviegoer will sit through a projection of a man walking. As genre theory progressed, it became possible to tell increasingly complex and powerful stories with the camera, allowing audiences to evolve alongside theory. Stanley Kubrick's *2001: A Space*

continued on page 55

continued from page 56

Odyssey expanded on the examination of movement by featuring a woman walking around a hatchway in a lunar shuttle for several minutes before the audience "got" the weightless environment. Now the same result is accomplished in seconds.

Redefining "genre" to refer to content instead of format will help validate the industry by establishing a solid artistic consensus

Evolving a Creative Grammar

Broad genre theory should be a component of academia. The next generation of developers will be the first educated at accredited game-studies institutions, and they will place considerable value on the benefits of pure scholarship. Curricula must include both technical and theory studies. The vast majority of non-production film classes are devoted to genre theory, typically focused on specific genre analysis or the examination of major directors. Students of game development would find value in a study of survival-horror's common structural elements, or a course on the "playing God" experience in Meier and Molyneux games.

There is already an enormous body of genre theory in film and literature. Game developers and scholars can use this material to great effect as groundwork for the development of a unique genre vocabulary for games. It is also necessary to break the habit of using the word "genre" to define game style. Nomenclature performing double duty in this capacity is ultimately just confusing.

Benefits of Genre Study

With the birth of a shared vocabulary in genre theory, the industry will gain a stronger defense against opponents of the medium. It is considerably more difficult to attack something possessing a solid artistic consensus. Developers, meanwhile, will have access to the tools and understanding necessary to produce thematically innovative games. Audience understanding of game genre will allow users to more effectively determine those genres that resonate most with them, which may help reduce high return rates.

There is no doubt that gaming lacks a complete language of its own. What it has is largely borrowed from the cinema, but film has no language for interactivity. Proper and distinct application of genre theory to game development and game studies means better games. The clumsy melodrama of early cinema is laughable compared to the slick productions of today, an improvement that stems from potent genre theory. Considering how thematically impressive videogames are even when they don't have genre theory, their narrative power when they do will be staggering.

MATTHEW SAKEY | *Matthew is a professional writer, designer, and gaming consultant, working with developers to leverage game technology for electronic learning. He writes the monthly Culture Clash column at www.igda.org and contributes to various gaming sites. Contact him at matthewsakey@comcast.net.*