G A M E   D E V E L O P E R   M A G A Z I N E

MAY 2003

UBM
Tech

GAME PLAN

LETTER FROM THE EDITOR

What's Good for the Goose . . .

I got back from GDC not too long ago, physically exhausted but mentally energized as always. Seeing the development industry face-to-face every year causes me to reflect on its state of diversity, as it's hard not to when you're swimming upstream in a rushing river of primarily male, white faces in their 20s and 30s.

To be honest, I'm never quite sure how to react when people bring up the fact that the editor-in-chief of a confidently eponymous industry publication for game developers is a woman. I at least get some entertainment value out of the people who begin a comment on this fact before realizing they failed to secure an intelligent-sounding or at least nonoffensive conclusion to the comment. And believe me, I'm not easily offended. But I do have a sense of humor at least, which I reserve for the well-meaning.

It's no secret that the short history of game development carries a long tradition of not having many women in it. Nonetheless, I'm encouraged for the future by seeing more women in more roles year over year at GDC, from speakers to attendees to volunteers to students, as I think many people in the industry are. I'm even more encouraged by the active efforts of the IGDA's results-oriented Women in Game Development Committee, as well as companies such as Microsoft, who sponsor an annual event at GDC called "Women Celebrating Women in Gaming."

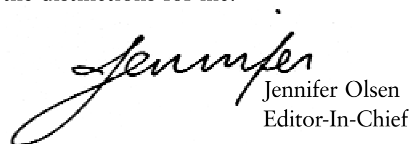
More and more people from all backgrounds are taking an interest in the issue of gender diversity in game development. The magnitude of the issue can be overwhelming to those wondering whether and how they can make a difference for the better. For one thing, it's important to remember that there is no single "issue" about gender in game development, yet I often see discussions in newsgroups and at in-person gatherings devolve quickly when the tendency prevails to bond the big slabs of distinct, complex issues with the mortar of generalizations. Following are some issues that I like to keep distinct from each other for the purposes of organizing productive discourse and formulating solutions.

Lack of female-friendly product on the market and a dire lack of information on the female consumer market itself doesn't inspire women to play games or make them. This may be the single biggest frustration facing women who currently work in the game industry and anyone in the industry who wants to target a broader or girl-focused audience rather than the "traditional" audience of young males.

There's a huge challenge in attracting women to computer science study and work that is much bigger and older than the game development industry. Organizations like Women in Technology International (WITI) and the ACM have been grappling with this issue as it has existed in the broader technology sector for far longer than computers have been used to make bikini-clad volleyball players jiggle across TV screens.

It's extremely hard (not to mention dangerous) to make generalizations of any kind about women, be they consumers or developers. Not all women are into touchy-feely "empowerment" efforts, not all girls want to play social-based games, not all women game developers magically know what women game players want, and to top it all off, not all generalizations are entirely false.

While these challenges seem formidable, the upside is that solutions we find will benefit all involved with the industry, not just women. Developers will learn more about the nature of what they do when more different kinds of people are doing it. They will learn more about themselves when they start questioning their assumptions and stop projecting them on others. Carol Shaw, creator of RIVER RAID, has been quoted on the gender significance of her pioneering work as a game designer and programmer, "I really don't like to make a distinction; other people always made the distinctions for me."



Jennifer Olsen
Editor-In-Chief

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

Publisher
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL

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

ADVERTISING SALES

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

ADVERTISING PRODUCTION

Vice President, Manufacturing Bill Amstutz
Advertising Production Coordinator Kevin Chanel
Reprints Cindy Zauss t: 909.698.1780

GAMA NETWORK MARKETING

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

CIRCULATION

Game Developer is BPA approved

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

SUBSCRIPTION SERVICES

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

INTERNATIONAL LICENSING INFORMATION

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

CMP MEDIA MANAGEMENT

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

GamaNetwork

WWW.GAMANETWORK.COM

INDUSTRY WATCH

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

The Khronos Group grows. The Khronos Group, whose members participate in the development and deployment of OpenGL ES and OpenML applications, recently announced that Ericsson Mobile Platforms has joined the group as a promoting member. FueTrek Company and Mitsubishi Electric Corporation also have joined the group as contributing members.

GameSpy allies with Vivendi Universal. GameSpy Industries announced an agreement to supply technology and back-end services, including server, bandwidth, and reporting support, for several online games to be released by Vivendi Universal. The first two titles planned under this agreement (for which no financial terms were released), are Relic's HOMEWORLD 2 and Impressions Games' LORDS OF THE REALM 3.

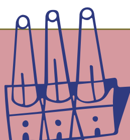
Impressions' LORDS OF THE REALM 3 will be one of the first Vivendi titles to share technology and services with GameSpy.

Sony moves production to China. Sony recently announced the company will be moving all Playstation 2 production to China within its next business year. The move, which was done to trim costs, follows Sony shifting half of its Playstation manufacturing to the Chinese factories of two Taiwanese electronics firms. Though there is some speculation that these moves have been made in part to set up production for the estimated 2005 release of the Playstation 3, Sony has said they do not know where the future console will be manufactured.

A gathering of developers perform ritual, create Skylab. Original members of Ritual Entertainment, led by co-founder Michael Hadwin, have joined forces with original members of Gathering of Developers, led by co-founder Rick Stults, to form Skylab Entertainment. Calling Austin, Tex., home, the company is working on their first unannounced title.

Acclaim target of class action lawsuit. The law firm of Cauley Geller Bowman Coates & Rudman filed a class action lawsuit against game publisher Acclaim for failure to disclose and/or accurately represent claims that, among other things, the company was engaged in aggressive sales practices; that Acclaim was suffering from decreased demands for its products; that the company had failed to meet revenue and earnings guidance; and that projections and forecasts concerning Acclaim were lacking in a reasonable basis at all times.

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



THE TOOLBOX

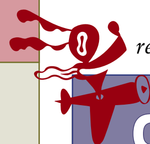
DEVELOPMENT SOFTWARE, HARDWARE, AND OTHER STUFF

Virtools releases Virtools Dev 2.5. Virtools has released the latest version of its flagship 3D development environment, Virtools Dev 2.5. The update features a new Virtools Scripting Language, a Schematic Editor, and an SDK. The company also announced the Virtools AI Pack, a behavior library for Dev 2.5. www.virtools.com

Criterion introduces several new products. Criterion launched four new products at GDC. RenderWare Physics is designed to enable developers to add real-time dynamic behavior to game objects and environments. RenderWare AI will provide direct, customizable access to AI tools. RenderWare Graphics 3.5 features an updated user interface. RenderWare Studio 1.2 includes such features as Build Process Tools, Enhanced Event Visualizer, and Spline and graph editing. www.renderware.com

Gamebryo is hatched. NDL unveiled Gamebryo, the successor to its NetImmerse 3D graphics toolkit and engine, at GDC. New features include an expandable tools architecture, pixel and vertex shader editing, and new animation tools. www.gamebryo.com

DTS offers PS2 SDK free for one year. DTS announced at GDC that they are offering a zero-fee license for use of its multichannel sound technology for Playstation 2 games. The license, which includes technical and marketing support, will be offered to all developers working on Playstation 2 games that are certified by DTS prior to April 1, 2004. www.dtsonline.com



UPCOMING EVENTS CALENDAR

GAME DEVELOPERS WORLD
BELLA CENTER
Copenhagen, Denmark
May 8–10, 2003
Cost: variable
www.gd-world.com

E3
LOS ANGELES CONVENTION CENTER
Los Angeles, Calif.
May 13–16, 2003
Cost: Free–$550
www.e3expo.com

ELSPA GAMES SUMMIT
RADISSON SAS, PORTMAN SQUARE
London, United Kingdom
June 17–18, 2003
Cost: Approx. $618–$2,245, + VAT
www.elspa.com

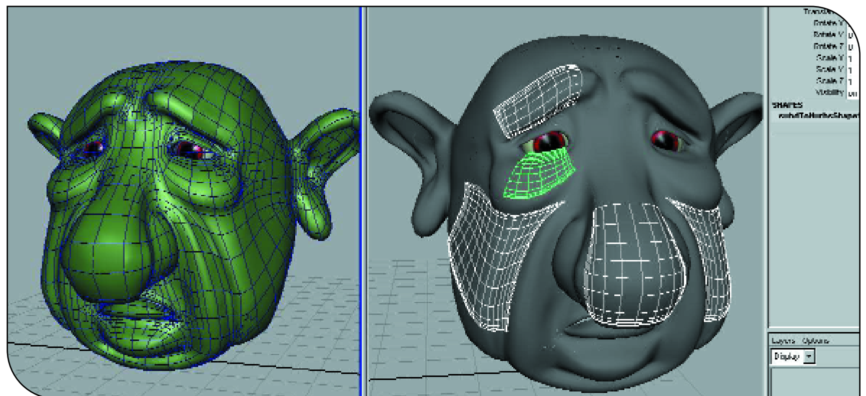
Alias|Wavefront's Maya 4.5

by spencer lindsay

Doing a "complete" review of a product with the scope of Maya 4.5 is a monumental task, so I considered my approach carefully. Given that my role at Rockstar San Diego is primarily one of a modeler and MEL guy, I decided to focus on what I know best and use the most, while also giving a circumspect look at the more intriguing offerings among this version of Maya's new features.

For an incremental update, Alias|Wavefront has presented in Maya 4.5 a fair number of advancements, including Fluid Effects, Smooth Proxy, a large number of API enhancements, new (and groovy) snapping tools, sub-d to NURBS conversion, interface changes that facilitate better workflow, and new polygon and beveling tools. But are these improvements significant enough to warrant an upgrade? Productivity and creativity are co-rulers of the game-art world, so I'll try to relate how these new features affected my workflow and my ability to translate my ideas to the screen.

Fluid effects. Based on my experience of using water plug-ins in the past, I tend to be intimidated by anything that has the words "fluid dynamics" in it. However, the Maya Fluid Effects package is actually very usable, with a great amount of depth if you need more out of it than the preset values (which are numerous). Our effects artists around the office are using this new functionality a lot and digging it.



Maya 4.5's ability to convert between subdivision surfaces and NURBS provides a new modeling workflow.

Workflow improvements. I've been using Alias|Wavefront's Bonus Games Package of plug-ins (available on the Alias|Wavefront web site for Maya 4.0.x) for a while now, so some of the workflow enhancements integrated into 4.5 are old hat to me, but they are nevertheless a huge step up in usability. Alias|Wavefront has bundled a large number of "patch scripts" into this version as well, including Poly Power Tools and the majority of the Bonus Games Package set, allowing you to do things like "Split Face," "Poke Face," and "Wedge Face," which can be combined in a large number of ways to give you some really great MEL scripts. They've also unhidden the Annotate command and put it in a pull-down menu. However, there are still some things missing, such as the excellent UV Texture Editor tools that come with the Bonus Games Package for 4.0.

Also among the new workflow improvements are the new snapping tools, which are very intuitive and easy to use. In particular, the "Snap Together" tool has proved extremely useful for the work we're doing right now, where lots of objects need to be placed with transforms relative to the surface of the landscape.

"Retain Component Spacing" is probably one of my most anticipated additions to Maya. Coming over to Maya from 3DS Max, I was frustrated with the fact that when I used the snap tools, it always snapped all the selected objects to the same transform direction. Now, with "Retain Component Spacing" switched on (the default now in 4.5), you don't have to cluster your components in order to snap them in relation to each other. It's a small but very welcome change.

Subdivision surfaces. Manipulating subdivision surfaces, or sub-d's, hasn't

SPENCER LINDSAY | *Spencer is a technical artist at Rockstar San Diego, where there are scary SUVs everywhere. He misses the redwoods. You can contact him at slindsay@rockstarsd.com.*

changed much in functionality, but Maya 4.5 adds the ability to convert between sub-d and NURBS, which is great if you do a lot of NURBS and sub-d modeling, which I don't. One thing I did notice was that during the conversion from NURBS back to sub-d, Maya tended to tessellate the edges of my isoparms unnecessarily, creating extra subdivisions in the new sub-d set. Happily, there are some added smoothing techniques which give you more control over the final polygonal model's density.

The best new thing in the subdivision-surface domain is the new "Smooth Proxy" tool, which is basically the same thing as 3DS Max's NURMS. A "smoothMesh" object is created with input connections from the "proxyMesh," so whatever large polygonal modifications you apply to the proxy are transferred down to the smoothed version. I played with this for hours and found it to be a very usable way to model polygons organically in Maya.

NURBS. Like subdivision surfaces, NURBS functionality remains largely the same. The most significant change is that all the functionality of the advanced modeling tools, like Booleans, Offset Surface, and the like, which were previously available in Maya Unlimited are now standard in Maya Complete.

API and MEL refinements. In Maya 4.5, Alias|Wavefront has continued to refine the MEL tools. Improvements include the ability to change the background colors in windows (PC only), the addition of lockNode/lock functionality, and the exposure and documentation of the "Annotate" command. The HTML help pages have been completely reformatted to be much more readable. However, version 4.5 still lacks an easy method of designing UIs in MEL, as well as a usable script editor.

Support. While Alias|Wavefront has made excellent progress with regard to feature additions and stability improvements, support is a relatively sore point. As with all products that make the price migration from high-end to consumer, Alias|Wavefront has stumbled in the support it provides for Maya. In my experience, although I can call or write e-mail and get a pretty fast response Monday through Friday between 3:00 a.m. to 8:00 p.m. Eastern time, I have experienced virtually zero weekend or late night support. As we all know, the game and film industries operate 24/7/365, and sometimes we need support right away during deadlines and milestones (I was fortunate enough to be experiencing both of these during my evaluation of Maya 4.5 for this review). With their growing customer base, Alias|Wavefront needs to invest some thought and effort into shoring up this weakness as soon as possible.

Last word. Maya 4.5 offers lots of useful additions to the workflow and modeling tools, and to the great relief of users the stability improves with every release. Clearly Alias|Wavefront has been listening to their game customers and their feature requests, a trend I certainly hope to see continue. If you can manage the significant hiccup Maya's rapidly growing user base has caused in the area of support, Maya 4.5 is a worthy upgrade.

MAYA 4.5

STATS

ALIAS|WAVEFRONT
Toronto, Ontario, Canada
(800) 447-2542 or (416) 362-9181
www.aliaswavefront.com

PRICE
$1,999 (MSRP)

SYSTEM REQUIREMENTS
Windows XP/2000: Pentium II or AMD Athlon with 128MB RAM (256MB recommended) and qualified graphics card; three-button mouse
Mac OS X, Linux, IRIX: see web site for requirements

PROS

1. New SmoothProxy tool kicks butt.
2. Lots of incremental workflow improvements add up.
3. Reformatted HTML documentation is more user friendly.

CONS

1. Customer support is slipping.
2. MEL still relatively ungainly.
3. Maya 5 already looming on the horizon at press time.

Real Sound Synthesis for Interactive Applications

by Perry R. Cook

reviewed by jeremy jessup

Real Sound Synthesis for Interactive Applications describes elementary and advanced techniques to simulate the audio components of dynamic systems using physics. While the book is not specifically directed toward game developers, the application to game development is clear. The book's organization is easily to follow through three sections detailing digital audio, sound modeling, and simulation of real-world instruments.

The first section (chapters 1–3) defines digital audio, compression, wave synthesis, and simple filtering techniques. These chapters serve as the foundation for what follows, defining common asset formats and techniques currently used in games.

The second section (chapters 4–8) introduces sound modeling through simplified physical systems, such as an ideal spring, and Fourier series equations. While an understanding of college physics and calculus is helpful (especially if you'd like to code these methods), the book doesn't require it or get bogged down in theory or mathematical proofs.

The last section (chapters 7–16) provides physics equations that allow for the simulation of real-world instruments (string instruments, tubes, and multidimensional objects). Each chapter describes a different system based on Fourier construction, filtering, and physics-based equations. It's the heart of the book and most interesting.

The clean organizational layout made it easy for me to refer back to previous sections when I felt the need. In many cases, however, I found the writing to be a little too condensed and wished for a paragraph describing a concept rather than the sentence provided. Cook does supply ref-

erences at the end of each chapter for those readers seeking additional detail.

The book also includes a CD containing audio samples of the topics discussed throughout the book. While reading the book, it was useful to be able to hear the point made or technique used in the text. The CD also contains Cook's sound synthesis toolkit and several examples of instruments highlighted in the last section.

Unfortunately, real-time sound synthesis in games currently has a limited place. Due to the complex calculations of Fourier series, fast digital signal processor chips are required to simulate the audio effects without impacting the rest of the game. Minimally, filters and other simple routines outlined in the book can be written for target hardware to accomplish specialized effects, but this is nothing radically new.

However, Cook's research in simulating audio is extremely exciting. During the calculation of an object's dynamic behavior (such as collision response, striking, falling, and moving), a minimal additional amount of time can be spent to determine the audio effects. According to Cook's findings, this amount is generally less than 5 percent of the total time required to simulate an object's physical behavior. Admittedly, these calculations are on the order of minutes versus milliseconds, but eventually Moore's law will catch up and simplifications will allow unparalleled audio effects in conjunction with physical simulation.

Developers and sound designers interested in the math and physics of creating real-time sounds should pick up this book. Those interested in a fascinating look at the mechanisms of dynamically producing sound might also want to give it a read, provided it's with the understanding that the direct applicability to games is at least few years away.

★★★ | *Real Sound Synthesis for Interactive Applications* | A. K. Peters | www.akpeters.com

Jeremy Jessup has been developing games professionally for nearly five years.

Whole Tomato's Visual Assist .NET 7.1 and 6.0

by noel llopis

If you're using Visual Studio, chances are you've heard of Visual Assist. For the three of you who haven't, it's a plug-in for Visual Studio (.NET or 6.0) intended to improve the everyday tasks of C++ programming.

Some of its features improve on things Visual Studio already supposedly does, like syntax coloring. Visual Assist takes it a step further and uses different colors for classes, variables, preprocessor macros, and class member functions. You can print the source code using those colors, or copy it to the clipboard in full color.

One of my favorite features is the ability to switch between an .H file and its corresponding .CPP file. Keeping the class declaration and the implementation in synch was never so easy; press a key and you're there. Visual Assist integrates very well with Visual Studio, allowing you to map any of its commands to keyboard shortcuts or toolbar buttons.

I often find myself searching for a specific function within a file, but I might not remember the exact name of that function. No problem, Visual Assist lets me bring up a list of all the functions in the current file, and with a click it jumps to that location. You can even sort the functions alphabetically.

Another indispensable feature is the ability to jump to the declaration or definition of any symbol under the cursor. Visual Studio claims to do that, but it often fails miserably, especially when dealing with symbols outside the current project. Visual Assist does a much better job, and only on rare occasions won't be able to find a symbol. They even finally implemented good namespace support. Visual Assist can also display the full declaration of functions under your cursor, including the type of all the parameters. It also provides very effective autocompletion for any symbol while you are typing.

As far as I'm concerned, the rest of the features are pure gravy: column delimiter marking, current scope display, auto insertion of parentheses and brackets, and syntax error underlining, to name a few. You can also turn off any feature you don't need.

So, what's not to like? My biggest complaint is the apparently rushed way in which new versions are released. Sometimes it seems that every week there is a new release coming out, and sometimes major bugs slip through. I spent a whole month getting lock-ups and crashes that I was blaming on Visual Studio, until I upgraded to the latest release of Visual Assist and it fixed all my problems. Needless to say, I was not particularly happy.

Installation wasn't totally smooth either. Maybe it's because I was upgrading from VC++ 6.0 to .NET, but at one point I had to fully uninstall Visual Assist and install it from scratch. Not a big deal, but I lost all my custom settings in the process. Some people have also reported incompatibility problems with other plug-ins. Whether that's a problem with Visual Assist or with the other plug-ins, I don't know, but you should definitely get the trial version and make sure you don't have any problems.

Their upgrade policy isn't particularly friendly. You have to pay an upgrade fee for each major version released, which is understandable, but the versions for .NET ($49 upgrade) and VC++ 6.0 ($29 upgrade) are considered two different products, so you'll be paying twice if you need to upgrade both versions. At least their pricing scheme is reasonable ($79 for either version new or $119 for a bundle).

All in all, Visual Assist is an indispensable aid for any C++ programmer using Visual Studio. But be warned: you'll feel withdrawal symptoms if you ever have to program without it.

★★★★ | Visual Assist .NET 7.1 and 6.0 | Whole Tomato Software | www.wholetomato.com

Noel Llopis is a software engineer at Day 1 Studios. Contact him at llopis@convexhull.com.

Been There, Done That

Tom Hall Revisits the Basics

Tom Hall wrote his first game, GOLD QUEST, on an Apple II+, in BASIC, in 1980. Since then, he has gone on to work for Softdisk, helped found id (COMMANDER KEEN, DOOM, QUAKE) in 1991, and then moved on to Ion Storm in 1997 (where he created ANACHRONOX). He has worked alongside industry notables John Romero and John Carmack for most of his career. In 2001 — along with Romero — Tom formed Monkeystone Games, where he currently serves as chief creative officer ("a fancy title for lead game designer," he concedes) concentrating his time on creating games for portable devices.

Tom Hall is re-examining the relationship between gameplay and technology on mobile devices.

With this issue of *Game Developer* highlighting mobile gaming, we gathered Tom's thoughts on this emerging medium and its impact on the game development industry.

Game Developer: Do you see the promise of mobile devices — and the delivery of information and entertainment through them — in danger of being overhyped, as the Internet was through the late '90s?

Tom Hall: Well, the Internet was useful already before it was hyped up — I was on it in college when the best I had at home was a 300 baud modem. Of course mobile devices are in danger of being overhyped. Everything is. But there are always technologies that are ripe for some application to come along and prove them indispensable. Mobile devices need that. Getting 20 cents off a burger because I walked past McDonald's with my SmartPhone is hardly a killer app. However, once everyone has a GPS on their phone, the equivalent of Mapquest becomes the killer app.

GD: In respect to graphics and programming limitations, what comparisons do you find between the early days of PC and console game development and now mobile games?

TH: You have to pull out your old-school knowledge, because you don't have that fast a system. It also makes you focus on gameplay, because you aren't going to wow them with thousands of polys. Also, it limits what kind of games you can do, as you can't fit a lot of text, and some devices don't allow you to press two buttons at once.

However, handheld devices are starting to get pretty impressive. You can have DOOM-level tech on some of them for sure, and — if you work really hard — QUAKE on a smaller set of them as well.

GD: How do you approach a project intended for mobile devices as opposed to a PC- or console-based title?

TH: I first have to think of a game that works in that resolution. Most phones are around 128×144 or so. So a lot of game types go out the window. And the app and the data have to fit in 200K. That takes care of a bunch more types of games. And then you have to be able to play it with a D-pad and a button, really. You can use the number keys, but if you're someone waiting for their plane, you don't want to get involved in an epic story or use complex controls. You can't press two buttons at once, so fighting games and shoot-em-ups like GALAGA or THUNDERFORCE are the wrong thing to try for.

However, we are now doing a 3D shooter for cell phones and it will use Bluetooth for multiplayer, so that's an exciting step. Whatever features the phone excels at, we try to take advantage of those. Shigeru Miyamoto gave a speech at the Game Developers Conference a few years ago, asking designers to study the technology they have at their disposal to get the most of it. That's what we try to do.

GD: You once said, "There's little room for doing true design in a technology-oriented company." How has this view changed since your days at id, Ion Storm, and now Monkeystone?

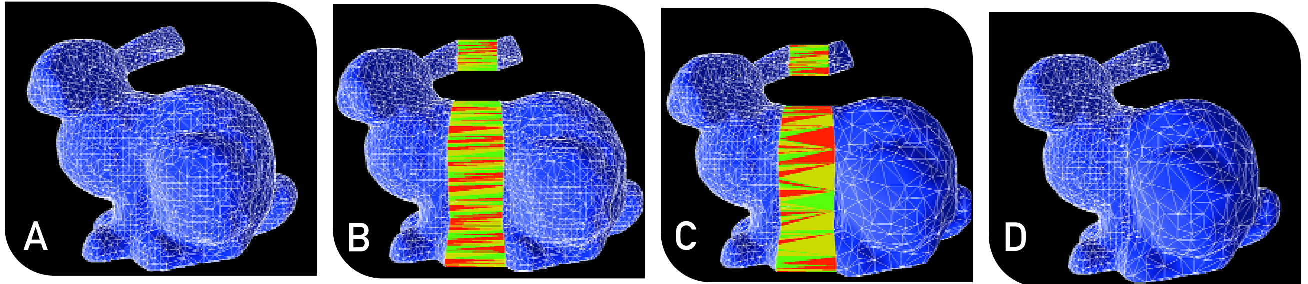
TH: That statement was simply my reaction to the focus at id, which was a great focus: "We have the best technology in the world, let's get it out there before anyone catches up to us." DOOM could get away with little content and no ending because it had amazing raw cool. But getting away with it, to the person wanting to have design innovations as well, was not the point. At that point, we'd made HOVERTANK ONE, CATACOMB 3D, WOLFENSTEIN 3D, and SPEAR OF DESTINY. I was ready to add to the bare-bones approach, to leave other people that much further in the dust. Romero felt the same way during QUAKE.

I absolutely stand by that statement. There's nothing wrong with having absolute focus — you get great things out of it — look at DOOM 3's technology. But with that technology, you could do a lot of insane, creative things. [id's] charter is to do more of what they've done, add minor stuff that shows off what the technology can do, and then let the licensees take the design risks. It makes total sense, but it isn't a situation where I could find my work rewarding and fun.

GD: What other portable devices do you think hold promise for the gaming industry?

TH: We're excited about the new Game Boy Advance SP, the SmartPhone, and the new Nokia N-Gage, especially. Finally, someone is making a phone for gamers.

Unified Rendering LOD Part 3



FIGURES 1A–D (from left to right). 1A: A 1,000-triangle Stanford bunny. 1B: The same bunny clipped in half with the halves joined together by a seam. 1C: The right half of the bunny has been passed through a detail reducer, decreasing its triangle count by a factor of 4. The seam has been cross-referenced and correctly preserves the manifold. 1D: The bunny halves, at differing LODs, have been moved back together and the seam is now drawn using the same shader as the rest of the bunny.

My goal up to this point with this series of columns on unified rendering level-of-detail (LOD) has been to create a general method of LOD management that works for a wide variety of geometry types. But so far I've only discussed the rendering of regularly sampled height fields. This month I'll start extending the system to triangle soups.

Recall that the basic algorithm works by dividing the input data into spatially localized blocks, then hierarchically combining those blocks. Because we were working with height fields, the vertices were spread mainly across two dimensions, so the array of blocks was two-dimensional. Since the input data was a grid of samples, dividing the data into blocks was easy, and building the initial seams between high-resolution blocks was also easy.

Now we want to allow the triangle soup to extend arbitrarily in any direction, so the array of blocks must be three-dimensional. Building the blocks and seams will also be more involved. But fortunately, we're already familiar with the necessary tools; we use them when constructing spatial organizations in a variety of triangle-processing algorithms.

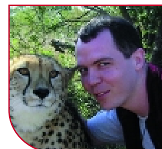
As an overview of what I'm discussing, see Figures 1a–d, where the Stanford bunny has been chopped in half and joined by a seam. The separate halves can change their detail levels; the seam tracks the changes and maintains the manifold. By repeating this operation with different planes, we can chop a mesh into a grid. (This month's sample code only does a single plane, since there are complications with edge and corner handling that I will discuss next month.)

Clipping or Binning?

We want to divide the input geometry into a three-dimensional grid of blocks. Since the triangles will not generally be aligned with block boundaries, the obvious idea is to clip the geometry using planes aligned with the faces of each block.

However, the goal of dividing the geometry into blocks was just to create groups of triangles, where each triangle is located near the other triangles in its group. Strictly speaking, there is no algorithmic requirement that prevents the blocks' bounding volumes from overlapping. So instead of clipping, we could divide the geometry as follows: (1) start with a set of empty "bins," each representing a cubic area of space; (2) for each input triangle, find the cube that contains its barycenter, and put the triangle into that bin; and (3) compute the smallest axis-aligned bounding volume for each bin by iterating over all the binned triangles. When we have done this, we'll have a set of rectangular volumes that are probably a little bigger than the initial cubes, so they overlap a little bit. (Pathologically large triangles might cause the bounding volumes to overlap significantly, but you can easily subdivide those triangles to eliminate the problem.)

Which of these methods, clipping or binning, should we implement? Unfortunately, clipping increases the total triangle count, which should be cause for concern. I chose clipping anyway, though, because even though non-overlapping bounding boxes were not a basic algorithmic requirement, they provide a greater interoperability with arbitrary rendering algorithms. Some algorithms require a strict rendering order for correctness, such as rendering back-to-front. If we're rendering grid squares that don't overlap, we can quickly achieve correct ordering of every triangle in the scene like this: (1) sort the triangles in each block



JONATHAN BLOW | *Jonathan is currently living in Austin, Tex. and not having any fun. E-mail him at jon@number-none.com.*

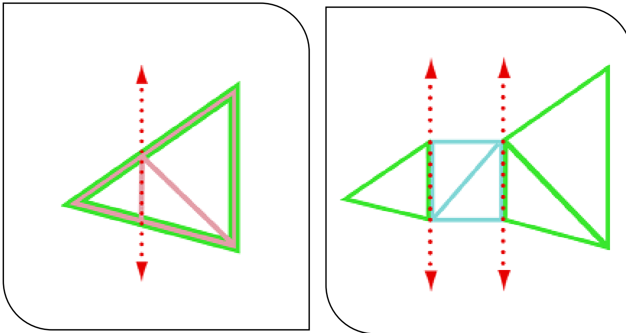


FIGURE 2A (left). A triangle (brown) has been clipped by a plane (red dashed line). The resulting convex polygons must be triangulated, yielding three triangles in this case (green). FIGURE 2B (right). We insert a degenerate quad (blue) to link the triangles on each half of the plane. The triangles have been artificially pulled apart here so that the quad is visible.

individually; (2) treating each block as an indivisible unit, sort the blocks by the distance from their center points to the camera; and (3) render the blocks in the sorted order. This sorting works because the grid squares are regions of space that result when you chop the space up with a bunch of planes. The applicable idea here is "Render everything on the same side of a plane as the viewpoint, then everything on the opposite side," which is the same idea that makes BSP-tree visibility work. A grid of cubes is isomorphic to a BSP tree; it's just stored in array form instead of tree form (and array form has a lot of advantages).

Now it's true that one of the basic goals of this algorithm is to enable the treatment of blocks as opaque entities, so that we can progress very quickly; if we need to sort the blocks, we lose much of that speed. But users of the LOD algorithm should ideally be free to make the judgment call themselves: whether to adopt an order-dependent rendering algorithm and accept slower running, or to run with unsorted blocks, getting higher triangle throughput, and using a different shading algorithm. If we don't clip, we make it much harder to sort the scene properly; we are effectively saying, "If you use this LOD algorithm, you won't be able to do rendering techniques X, Y, or Z without undue pain."

In fact I referred to an order-dependent rendering technique last month: the color-blending method of interpolating between LODs, used on DirectX 8 and earlier hardware. Rather than sorting the triangles in each block, the algorithm used the Z-buffer to effectively sort on a per-pixel level. This color-blending technique already tolerated some inaccuracy in order to run quickly: occluded triangles in the translucent layer for a block may or may not contribute color to the rendered scene, depending on the order the triangles are stored in the mesh. However, due to the way colors were being blended, the inaccuracies are small and hard to notice. I suspect that if we used binning and thus rendered the scene slightly more out-of-order, the resulting errors would be similarly subtle and the result would be acceptable. Just the same, though, one of my major goals has been to make the system maximally compatible with unforeseen game engine components. It seems like strict ordering is a potential trouble spot for cross-algorithm interactions, so I chose clipping.

Then again, maybe the set of conflicting rendering techniques is small and unimportant. If we decide not to support them, we get a simpler LOD system, because binning is simpler than clipping. A simpler LOD system is desirable so that when users need to modify it, their job will be easier. Really, this is a tough call to make.

Another reason I chose clipping in the end was that, with regard to seam building, clipping is a superset of binning: you need to solve the weird cases that happen with binning and do extra work on top of that to clip. So a clipping solution illustrates both cases, and if you wanted to simplify the preprocessing stage to use only binning, you could streamline the sample code instead of writing code from scratch.

Clipping and Triangle Count

The extra triangles created by clipping may not matter much. When we give the block to our mesh simplifier — assuming the simplifier works effectively — these triangles will mostly be seamlessly merged, so they'll have a small impact on the resulting triangle count and topology. Thus the time the extra triangles should matter most is in the highest-resolution blocks.

On fast, modern hardware, when geometry is finely tessellated, a relatively small number of triangles are created by clipping. To show why, let's once again look at a mesh built from a square grid of vertices. The number of triangles in the block is $2n^2$, where n is the number of samples on each side of the block. But the number of triangles added by clipping is approximately $4n$. So as we increase the resolution of our world mesh, the number of triangles added by clipping grows much more slowly than the number of triangles we actually render.

Building Seams

I recommend following the same strategy as last month when it comes to seams: create seams between the meshes at the highest resolution, then cross-reference them into the reduced meshes. The results are fully precomputed seams that you can render quickly.

For every line segment produced by clipping, we must create a degenerate quad to bridge any potential gap. Figure 2a shows a triangle that has been clipped by a plane, and Figure 2b shows the degenerate quad that links the triangles on each half of the plane. Because we're working in 3D now, the possibility exists that one of these segments will lie on a border between more than two blocks. If we employ an algorithm that generates seams between blocks based on which planes the segment touches, we could generate strange, spurious seam triangles.

There are some other factors of which we need to be cautious. When clipping geometry by a plane, we usually classify vertices as being in the plane's positive half-space, being in the negative

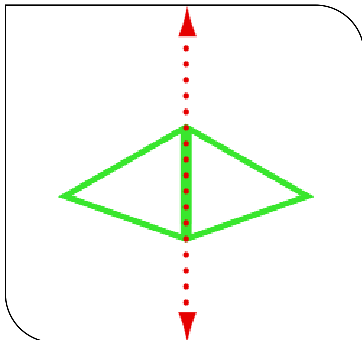


FIGURE 3. Two connected triangles straddling a plane.

half-space, or lying on the plane. We can end up in a situation where two connected triangles straddle the plane as Figure 3 shows. Even though clipping doesn't create new geometry in this case, we need to detect it and add the appropriate seam.

In this month's sample code, which you can download from www.gdmag.com, I wanted to solve all these problems while trying to keep things simple. The approach I take is clipping each block in isolation, then matching up seams between the blocks after the clipping phase is complete. Whenever I create a new edge via clipping, or detect an edge coplanar to one of the clipping planes, I add it to a list of potential seam edges. When all the blocks are clipped, I look at neighboring blocks and match up edges, creating degenerate seam quads between them. This two-pass approach is also important for out-of-core clipping, which may be necessary for large input geometry, so it's good to have it already built into the core algorithm.

When matching up the seam edges between blocks, we need to ensure that we don't attach the wrong edges to each other. There might be ambiguity due to multiple vertices located in the same positions, or due to floating-point calculations coming out slightly differently when edges are clipped multiple times. (Ideally, such floating-point problems don't happen if you're careful in the code that computes clipped coordinates. However, the current state of FPU management in Windows game software is dicey, so extra care is warranted.) To eliminate all ambiguity, I identify each edge by its two endpoint vertices, and each of those vertices is identified by three integers: the two indices of the vertices in the input mesh that comprised the segment that was clipped to yield this vertex, and the index of the clipping plane that clipped that segment. Basically, we have a unique integer method of identifying each clipped edge, and we don't rely at all on the floating-point output of the clipping system to match them up.

In last month's column, I showed that, in the height-field case, holes will appear in the mesh between the corners of blocks at varying LODs. An analogous problem happens in 3D, and we need to analyze that. Also, I haven't discussed methods of deciding which LOD of a given block to render. I'll get to both of those next month.

Scoring the Unknown
A Case Study in Building an Interactive Soundtrack

The growing sophistication of videogames has demanded soundtracks that are both dynamic and intimately linked with gameplay. Unlike film or television, in which characters, locations, and plot are predetermined, these elements are subject to rapid change in the realm of gaming. So how is a composer supposed to score a videogame keeping in mind that music elicits emotions in players and therefore has a direct impact on how the gameplay unravels?

In the summer of 2002, I began writing music for Naughty Dog's forthcoming JAK II. It became clear early on that the gameplay was going to be much more intense than its more exploration-based predecessor. In addition to an increased amount of opponents, a number of accessories were now available to the player at any given time. The soundtrack needed to address these new gameplay elements.

The challenge. The environment of JAK II is made up of several expansive locations designed for a free-roaming, nonlinear gaming experience. Each location has its own look and specific set of tasks. In addition, a variety of opponents enter the game at varying intervals. Also, accessories are integral to JAK II's gameplay, and their use enables players to complete tasks and defend themselves. The urgency of accessory use necessitates an immediate entrance and exit of its assigned musical layer.

The allotted memory permitted enough space for a looped MIDI sequence, which provided a bed that captured the mood of each location, and 20 extra MIDI channels to address accessory use.

The large volume of MIDI instruments necessitated the main level music bed to be simply constructed and memory-efficient enough to allow for these extra channels. The technical and creative challenge was composing the extra musical layers in such a way that their instantaneous, gameplay-dictated entrances and exits would be both effective and organic to the main music bed.

JAK II, slated to hit shelves in late 2003, will feature a more dynamic experience than its exploration-based predecessor.

Each appended musical layer needed to be composed to run the entire length of the MIDI sequence and remain muted until the player chose to use a particular accessory. Layering different percussion parts initially seemed like a good solution. The non-tonal quality of percussion didn't interfere with but rather enhanced the main music bed and seemed to sound natural when entering and exiting. Unfortunately, layering exclusively percussion-oriented parts didn't allow for enough variation to differentiate one accessory properly from the next.

The solution. We decided to use staccato-pitched instruments. The percussive nature of the parts written for these instruments retained the desired continuous driving effect while still following the chord progression of the main music bed. The relatively simple orchestration of the main music allowed enough space for the extra layers to play within a fairly uncluttered environment. Thankfully, the gameplay only allowed for one accessory to be used at a time. This also kept the soundtrack from getting too busy.

The issue of how the extra MIDI tracks entered and exited was dealt with as such. For example, if a player decides to utilize the hoverboard accessory, the accompanying hoverboard instruments are unmuted concurrently with that decision. The extra musical layers were composed as repeated one-to two-bar patterns. These MIDI channels were programmed to mute or unmute on any given downbeat without sounding too unnatural. To avoid possible auditory fatigue from the repetitiveness of these instruments, we thinned out their frequencies. This gave them a lighter, more transparent quality, which helped them blend into the overall music mix. As the extra layers entered and exited the main music bed, the soundtrack was finally starting to feel dynamic and scored to gameplay.

After a couple months of writing music for the game, Naughty Dog's sound guru Bruce Swanson and I stumbled upon a further step toward a truly interactive score: silence. Instead of building layer upon layer within a looped music bed which played the entire time a player visited a given location, we discovered that stopping the music periodically and bringing it back in at appropriate moments profoundly increased its effectiveness. By punctuating key actions and scenarios, the score is given a purpose beyond a relentless musical backdrop.

Composing music for multiple gameplay-dictated scenarios often felt like scoring each scene of a film or television show over and over again; a bizarre, modular, creative, and technical puzzle. However, the end result is more successful in drawing the player farther into the realm of the game, a goal worthy of achieving.

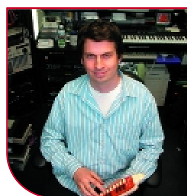


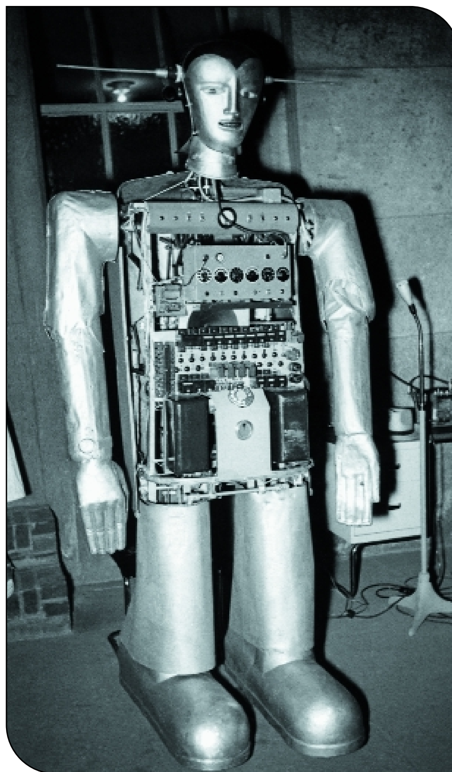
photo by Mr. Bonzai ©2003

JOSH MANCELL | *Josh began composing for Mutato Muzika studios in 1992 and has scored such interactive titles as INTERSTATE '82, JOHNNY MNEMONIC, CRASH BANDICOOT (first four), JAK AND DAXTER and its forthcoming sequel, JAK II, as well as television spots and radio commercials. Josh received an Emmy nomination in 2002 for his music on the* Clifford the Big Red Dog *television series.*

The Price of Progress

The game industry is a strange one with which to be involved. There aren't that many industries that will pay well-educated professionals to argue the merits of giant killer robots versus mutant zombie wolfmen. As far as I'm aware, (I'm sure you'll correct me if I'm wrong), no other industry in the world that has yearly revenue as great as ours relies on such a small number of 20- and 30-year-olds to create its product.

Of course, the "game industry" includes by association advertising executives, journalists, recruitment agencies, publishing lords and minions, and countless others, but regardless of how you look at it, without those of us who makes the games, there would be nothing to promote. So this month I'd like to focus on some of the changes that are taking place in the game industry from an artist's point of view, and how they affect the way we all work.



Changes for Developers

As far as the changes that are affecting game companies themselves, team size comes to mind first. The size of teams working on each project has increased from the artist-programmer pairings of the 1980s to the gargantuan extremes of the massive Japanese franchise teams we've seen in the last few years. We find the "average" developer putting between 20 and 30 people on a project that is aiming to be AAA, or if time is short and budgets are high, up to (and beyond) 50 people can comprise a mid-sized team.

While work practices differ considerably between companies, the sheer volume of art assets that need to be created and managed, all within a tight schedule, has meant that art teams are becoming more compartmentalized. The production pipeline is now often divided into specialist subcategories that define 2D texture artists, character modelers, environmental modelers, concept artists, and other specializations.

Additionally, art teams in many cases now include art directors or art leads, who inevitably spend the majority of their time managing rather than contributing assets. This is not to suggest that this layer of personnel is necessarily a bad thing, but the more development teams grow, the more middle management will be required, further inflating team size, and increasing wages and project costs.

Also, we are seeing an increase in the number of managerial roles that involve working directly with the art team. As the industry continues to mature, senior managers are likely to migrate toward the role of manager.

Do these kinds of changes lead to more or less autonomy for individual artists? On the one hand, larger projects run more smoothly if artists are proactive in their work, able to take charge of their particular tasks without the need for constant supervision. On the other hand, with a larger team and what ends up as being less actual input from each individual artist, creative independence can be seen as counterproductive to an overall aesthetic, and in many cases, autonomy is not part of the job description.

As artists find their roles becoming more specialized, the danger of accumulating an overly specific skill set becomes greater. At the end of a project, or with the demise of a company, it may become harder to find appropriate work elsewhere unless your specific skills are highly in demand.

These movements toward large budgets and huge teams put pressure on independent developers, and the future of independents is called into question. Thus, art direction in today's game indus-

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

try has to also reflect the reality of development pressure, where team size and resources must be considered alongside artistic vision, especially where independent developers are concerned.

Changes for the Publishers

It's easy as a developer to look at the industry's woes and point the finger of blame squarely at the publishers. Whether or not that's fair, it's certain that publishers are in the ascendancy as far as the balance of power goes.

As the principal holders of the development budget purse strings, in most cases publishers control what games get made, when, and by whom. In recent years we've seen a significant shift away from original games towards the "safer" ground of licenses and sequels. It's far less risky for publishers now to pay for the rights to make a game out of the next original potential blockbuster movie, than to invest in the next original potential blockbuster game. Currently, something like the top 5 percent of games pays for all the others that don't quite make it, and publishers are always looking to score the next multi-million-dollar seller that they can turn into a long-standing franchise.

Is the situation likely to change for the better? As it stands, a publisher Super League has developed, with four or five hugely powerful publishers on top, and the other weaker publishers relegated to the lower ranks. As with other industries, consolidation and mergers seem inevitable as the largest companies aim to squeeze out the opposition by becoming the biggest fish in the pond. The trend benefits neither developers nor consumers, as the less real competition there is between publishers, the fewer opportunities there will be for diversity and originality in games.

Changes in the Market

As an industry, we are continuing to sell more games each year. So who buys games these days?

It depends on what kinds of games,

The tremendous popularity of Maxis' THE SIMS among women has shifted the gamer demographic.

time of year, and platform you're talking about, but there are two broad consumer trends of which developers should be aware.

First, in the last five years, there has been a reported increase in the number of female game players, attributed primarily to both the rise of online gaming and the phenomenon of THE SIMS. Mainstream games specifically targeting the emerging "female market" haven't exactly flooded the shelves. This indicates a level of industry skepticism resulting from past failed efforts to offer female-targeted games, as well as a reluctance from publishers to invest significantly in something that strays away from the safer ground to

As much diversity as exists in the game industry, such "typical" game character as the buxom female from DEAD OR ALIVE XTREME BEACH VOLLEYBALL (left) and superheroes like SPIDER-MAN (right), still pervade the consumer consciousness.

which they are accustomed.

Second, comic book heroes with abs of steel and women with breasts that could easily double as life rafts still grace the front of every game magazine on the shelf. The 14-year-old boy (or at least the 14-year-old-boy mentality) still seems to be a dominant force in our industry.

While I am not suggesting that this is a problem, as these types of game form a solid part of our output, I think that games are going to struggle against the idea that they have no more to offer than big guns and leather-bound cleavage as long as the public at large is presented with these images more than any others.

As artists, is it our responsibility to expand the range of imagery on offer to help broaden the scope of the next generation of games? That's up to you and your company to decide, but while the stereotypes will always have their place, a maturing consumer base will be best served by products that provide entertainment on a whole range of levels.

Overall, it looks like the market is developing, but opportunities are also being squandered when they are seen as a departure from traditional, previously successful gaming genres.

Changes in Technology

It would be a huge understatement to say that over the last 10 years the technology associated with videogames has improved significantly. Technology is always racing forward, especially where electronics are concerned, but as the game industry consistently grew in size and revenue (faster than many other areas of high-tech), investment soared. We now find ourselves with gaming systems that two decades ago would have been multi-million-dollar supercomputers.

On the one hand, we have the aforementioned creative freedom and the

ability to bring our ideas more easily to the screen. The software we now have available to us makes our jobs as an artists far easier, while providing us with tools that can give us room to experiment in ways that previously we may never have imagined. All this is great, and I can't see many people opting to return to the days of 8-bit color and textures no larger than 32×32.

On the other hand, is there a price to this progress? Without sounding disingenuous, it can be argued that technology moving so fast has added significantly to the difficulties of developing (and releasing) a successful game.

The race to deliver a game that remains at the cutting edge visually is exceptionally difficult to win, and specifically with PC gaming, the choice is either to target the top 5 percent of systems and push the boat right out, or to aim at lower-spec PCs, which obviously includes more consumers but restricts what you can do. Determining an answer to this question depends largely on your product and to whom it is most likely to appeal. But with graphics cards settling into a six-month release cycle and faster processors constantly being churned out, a two- or three-year project has to fight hard to remain at the top of the performance heap.

Consoles alleviate a certain amount of hardware pressure, but the three consoles currently on offer are significantly different in terms of performance and market placement, leaving a developer with several options. One can choose between specializing in a single platform, exploiting all of its strong points to deliver a more impressive game (particularly visually), or trying to spread their project across all formats.

The all-format approach has the obvious advantage of the widest potential market, but unless resources are significant, cross-platform development can end up being a project killer, as no single format gets the attention it needs, so all versions suffer. Producing art assets across a number of platforms can be frustrating where different technical specifications have an enormous impact on what can be used. While it is fair to say

The proliferation of consoles creates mass-market opportunities for developers but increases competition greatly for smaller studios.

that the number of triangles that each console can push is theoretically massive, memory issues have a constraining effect on textures as well as the different ways in which each console achieves special effects, for example.

Targeting a specific console is much tidier, but limits your game to a smaller set of potential customers.

Throw into the mix the fact that the PS2 is the oldest and least powerful of the consoles, but also happens to be the one that has an installed base that towers way above that of its competitors. So what choice does a developer really have?

The answer to that question probably lies with the developer itself. A large-scale, multi-platform project is an unrealistic goal for any developer without significant resources. The attraction of the widest possible market is all well and good, but a finished game is infinitely better than a half-finished one, even if the half-finished one is in fact multi-platform. The crunch often comes when a publisher demands the widest possible release imaginable and the only sensible outcome has to be one that is realistic, otherwise chances are that no-one will be happy in the long run.

In addition to all of this, we are told that the "console lifespan" is now around five years, so if our AAA game is going to take up to three years (and that is not over-generous as an estimate), we only have time for one full production cycle before a new raft of consoles hit the shelf.

While this seems to be bad news on the face of it, in a sense, it serves to focus effort on what it takes to make a successful title. In many cases, an initial game, especially if it happens to use an original IP can be a solid foundation for a lucrative franchise. Sequels are the easiest way to generate income if the first game in the series did well, and from a production point of view, leveraging the technology from the original game is usually the way to go, with the focus being on minor improvements and, of course, a whole boat load of new art.

Sequels, such as MYST III (right, shown with the original MYST, left) can be an effective way to leverage income from a successful first game and save on the development costs of new technology.

The juggernaut that the games industry has become will continue to surge forward, controlled for the most part by those with the cash, fueled by a potent brew of technology and marketing. In the end, few of us will have any say in where it takes us, but for those of us who want to stick around, the best advice is to strap in tightly and try to enjoy what is bound to be a bumpy ride.

ralph barbagallo

Mobile Game Development on the Open Road:

Sun's J2ME Profile Takes a Major Step Forward with MIDP 2.0

Illustration by Glenn Hanson

Much has been said about Java 2 Micro Edition (J2ME) being a "standardized" API for mobile applications. However, Sun's J2ME profile for mobile phones, the Mobile Information Device Profile (MIDP), initially failed to address game developers' needs with its extremely sparse feature list. This list included a lack of standardized support for pixel transparency, network communication restricted to only HTTP, an inflexible GUI, and a lack of support for sound and music. To overcome these shortcomings, handset manufacturers introduced custom extensions to the basic J2ME/MIDP specification. Soon, it was as if every J2ME handset was its own platform. A MIDP "standard" was relatively meaningless, as you had to cater your code to every handset's nuances and special extensions. Those that stuck to basic MIDP for maximum compatibility were left with a bland and featureless gaming experience.

There are two major standards in the mobile gaming marketplace: J2ME and BREW. BREW (Binary Runtime Environment for Wireless), Qualcomm's wireless application platform, provides a lower-level C/C++ API for writing mobile applications on BREW handsets. Although implementations of J2ME Virtual Machines have been written on top of BREW and despite the potential symbiotic relationship of the two later down the road, BREW exists largely as J2ME's competitor. Although not perfect, BREW filled many of the gaps that MIDP lacked — including support for pixel transparency, TCP/IP sockets, and MIDI music from the SDK's first release.

Handsets supporting advanced features such as 16-bit graphics, multiple key-press, and advanced audio mixing are rolling out at an increasing pace. In turn, Qualcomm has released several new versions of BREW. The latest edition, BREW 2.0, introduces many game-specific features, including a built-in sprite and tile engine. However, handsets with BREW 2.0 installed on them have yet to appear commercially and may not do so until the fall.

Although it has taken them a while to act, Sun has not been ignorant of these developments. Taking suggestions from major game developers and publishers, among others, they have formed the MIDP 2.0 specification. In addition to features suited for all sorts of mobile applications, MIDP 2.0 contains a new Game API that addresses a lot of MIDP 1.0's shortcomings. Combined with a horde of new nongaming features, J2ME now rivals BREW's feature-richness. Combined with the relatively inexpensive and simplified Java development process, MIDP 2.0 may become a serious threat to Qualcomm's more gaming-friendly development environment.

The GameCanvas

In MIDP 1.0, all screen painting and input operations were handled by the `Canvas` class. However, the `Canvas` object could only receive one key event at a time, making it impossible to hold down two keys simultaneously — for instance, running while shooting. The new `GameCanvas` object is subclassed from `Canvas` and includes new features, including the ability to poll the state of the keypad with the `getKeyStates()` method. This function returns an integer that contains bitflags for all the currently pressed keys. In many cases, the ability to detect multiple keys is a hardware issue, so despite MIDP 2.0's support for multiple keys, it's still up to hardware manufacturers to include this feature in their handsets.

WHAT'S NEW IN MIDP 2.0

- `GameCanvas:` The new `Canvas` class allows multiple key-press and partial-screen repaints.
- `Sprite:` This new class handles the drawing, animation, movement, and collision.
- `TiledLayer:` This class represents a tile map. It handles the scrolling, animation, and collision detection of tile worlds.
- The drawing order of sprites and tile layers is managed by an easy-to-use `LayerManager` class.
- Signed MIDlets now allow you to use APIs which may not adhere to the sandbox security model.
- MIDP 2.0 now has sound and video playback as standard features. It is possible to play MIDI music as well as MPEG video streams.
- Over-The-Air Provisioning is now a standard part of MIDP. This means the interface to installing and downloading your MIDlet will be consistent between handsets.
- Vastly improved messaging support allows for TCP, UDP, HTTP, HTTPS, and SMS communication.

The `GameCanvas` class also allows the use of an off-screen buffer for double buffering. The `flushGraphics()` method will then transfer this off-screen buffer to the main display. It's also possible to use an overloaded version of `flushGraphics()` to update a smaller region of the screen for dirty-rectangle optimizations. Depending on the hardware, this may dramatically increase your performance for games that do not have a lot of on-screen motion or animation.

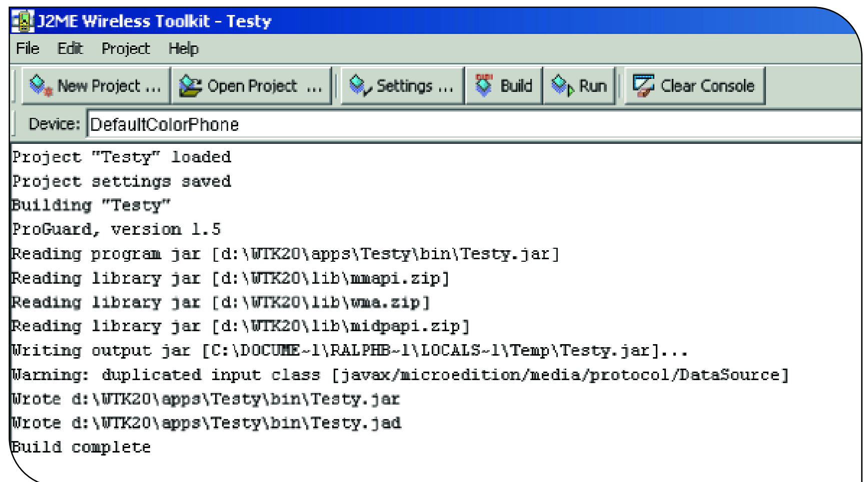
Graphics Enhancements

MIDP 2.0 has a series of graphics enhancements that both fix old problems and add new features game developers have been pining for during

RALPH BARBAGALLO | *Ralph runs Flarb Development (www.flarb.com), a Los Angeles–based game developer that has created leading wireless games for various publishers and carriers. Prior to this, Ralph worked at various studios, including Ion Storm, where he was a programmer on the RPG* ANACHRONOX. *Ralph has a B.S. in computer science from the University of Massachusetts at Lowell and is the author of the book* Wireless Game Development in C/C++ with BREW *(Wordware Publishing, 2003).*

the past few years of MIDP 1.0's reign. These include native sprite and tile support which, combined with hardware that fully supports these features, may bring us Game Boy–quality visuals and smoothness on J2ME handsets in the near future.

Personally, I would prefer a simple but fast hardware-accelerated blitting system instead of forcing developers to use a restrictive sprite and tile architecture that may or may not be suited for the game's needs. Perhaps Sun took it a bit too literally when developers asked for sprites and tiles. Whatever the case, the sprite and tile systems provided in MIDP 2.0 are leagues ahead of the graphics support in MIDP 1.0.



The new and improved KToolbar IDE.

Sprites

Both sprites and tiles are based on what is called a `Layer`. A `Layer` is an abstract base class that represents any visual element. The `Sprite` object is subclassed from `Layer`. The `Sprite` class contains all the functionality for drawing, moving, and animating sprite graphics.

MIDP 2.0 still uses PNG as the basic file format for all graphics. To create a `Sprite`, you pass a reference to an `Image` object created from a PNG to the constructor. To conserve memory, multiple sprites can be created from the same `Image`. If you want to draw the `Sprite`, call `Sprite`'s `paint()` member; however, the `Sprite` class has a lot more functionality beyond this basic usage.

Because the `Sprite` is derived from the `Layer` class, it inherits the basic size and position functionality of the abstract `Layer` object. Thus, by using the `move()` member, you can move the `Sprite` through a world-coordinate system by an arbitrary number of units. You can also explicitly set the position using `setPosition()`, as well as hide the sprite using `setVisible()` and passing false as an argument.

The `Sprite` allows a custom "reference pixel" to be set — which is the origin of the sprite. Back in the 1.0 days, you had the anchor argument in `Graphic`'s `drawImage` method to control the origin of a drawn image. Through the use of the `setRefPixelPosition()` function in the `Sprite` object, this origin can now be set to any arbitrary pixel in the image. This point can be continually reset for animations or other instances that require the origin to be modified over time.

Another interesting feature of the `Sprite` class is the ability to apply transforms. Through the `setTransform()` method, you can flip, rotate, or mirror any sprite in 90-degree increments. The ability to flip sprites is essential for developers trying to conserve space in their MIDlet JAR files. Before MIDP 2.0, you had to have a separate image of the sprite facing in each direction. Now, it's a matter of flipping or otherwise transforming the sprite to the desired facing. This can crunch down your graphics usage dramatically, depending on the artwork of the game. All sprite transforms are relative to the reference point, making it possible to "orbit" a sprite around any arbitrary point in the image. This ability often ends up being an annoyance, as in many cases you simply want a horizontal or vertical flip regardless of the reference point. In some cases, this action may make the reference point system somewhat useless.

The `Sprite` class also contains functionality for animation. In order to have an animated sprite, you must use a PNG image that contains all the frames of animation. These frames must all be of equal size and can be in the form of a wide strip, narrow strip, or square layout. The frames are automatically numbered in row-major order. When an animated `Sprite` is created, the width and height of each frame must be specified either in the constructor or via the `setImage()` method.

Individual frames can be displayed by calling the `Sprite`'s `setFrame()` function before painting it. However, frame sequences can give you more complicated animation control. A frame sequence is an array of integers with each entry representing a frame to display at that point in the animation. These frames can then be advanced or backed up using the `nextFrame()` and `prevFrame()` methods, as well as explicitly referenced using `setFrame()`. The `setFrame()` method takes an argument which is an index into the frame sequence. The default frame sequence is a list from 0 to the number of frames in the image. Thus, `setFrame()` still works even if you haven't set a frame sequence; frame sequence entry 0 references the `Image`'s frame 0, and so on.

Finally, MIDP 2.0's `Sprite` class contains some collision detection functionality. First, each `Sprite` can have a unique collision rectangle. By default this is the size of the frame or `Image`, however it can be explicitly set using the `defineCollisionRectangle()` function. Where the sprite is in the world is determined by the parent `Layer` class's position functions.

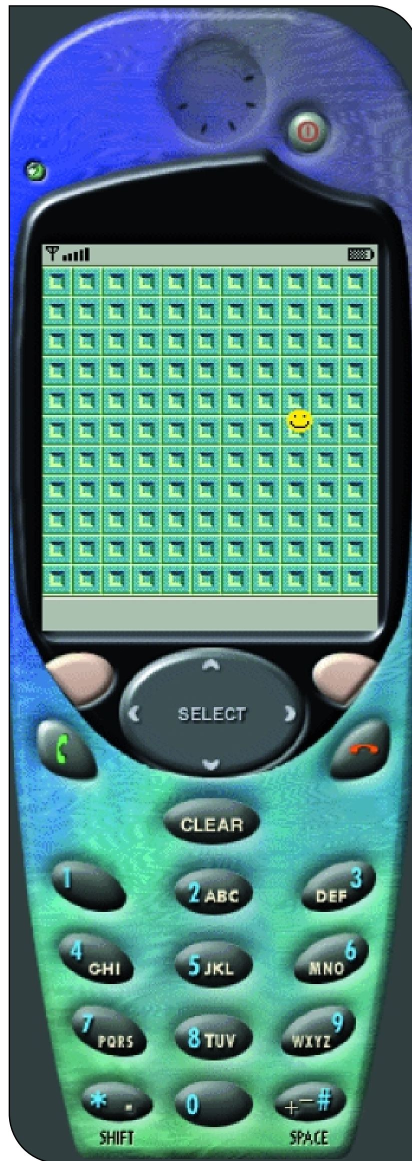
Now you can check if the Sprite has collided with other Sprite, tiles, or even Images. Inside the Sprite object, there are three overloaded versions of the collidesWith() method, each one used for one of these previously mentioned cases. Each takes a Boolean argument, pixelLevel, which if set to “true” will perform a pixel-accurate collision test. Otherwise, the collisionRectangle, tile width, or Image dimensions will be used to check for intersections.

Pixel transparency, which was not part of the formal specification, was another major issue with MIDP 1.0. Handset manufacturers either had custom extensions for transparent pixels in Images, or did not support them at all. For instance, in the case of Motorola’s VM, they supported transparency in the PNG palette using the tRNS block. With Siemens, you had to provide a 1-bit mask and use their custom API to have transparent pixels in sprites. So far, Sun’s emulator supports transparency palettes in PNGs much like Motorola’s J2ME handsets. It remains to be seen whether this will be standardized among all MIDP 2.0 implementations on actual devices.

Tiles

Next up on the list of graphic enhancements is support for tile backgrounds. MIDP 2.0 achieves this through use of the TiledLayer class. Much like the Sprite, this class is derived from Layer and thus contains all of Layer’s functionality in addition to its own. TiledLayer contains functionality not only for loading tile set graphics and displaying scrolling tile maps, but also for animating individual tiles for advanced graphic effects.

In many respects the TiledLayer object behaves much like a Sprite. One of the arguments to TiledLayer’s constructor is an Image object, created by loading a PNG containing all of the tiles used for the map. The tiles are basically the same as sprite animation frames; they must be all of equal size, and can either be delivered in tall strips, wide strips, or a large rectangle. The actual tile map is set by calling the setCell() function. Here you pass the X and Y locations of the cell and the tile index you wish to set. This tile index references the tiles in the Image much like an animation sequence does frames of a Sprite. To scroll the map, you call move() or setPosition() as defined in the Layer class. Drawing the tile map is as easy as calling paint().



THE MIDP 2.0 EMULATOR IN ACTION. This example is running at a glorious two frames per second with a single sprite and a simple tilebackground layer.

The Layer Manager

While it’s possible to call paint() manually on all of your Sprites and TiledLayers, MIDP 2.0 provides the LayerManager class, a class that handles some of the more mundane details of drawing Layer objects.

The LayerManager is an object that maintains a list of Sprite and TiledLayer objects, and draws them in their current position in the order that you add them in. You can adjust the Z-order of sprites and tile maps for drawing with different priorities as well. Those familiar with Game Boy Advance programming can think of it is as the MIDP 2.0 equivalent of the OAM table, in that it maintains a list of active sprites and tile backgrounds and draws them in the order you define.

Using LayerManager is easy. Construct the object, and then either add Layers by calling append() or explicitly set the Z-order for them by calling insert() to specify the position in the drawing list. Calling LayerManager’s paint() method draws all the Sprites and TiledLayers in the order they are currently stored in the drawing list. Using setViewWindow(), you can also alter the size of the viewport. This may be useful for reserving some screen real estate for score display or other HUD information.

Mobile Media API

Included in MIDP 2.0 is the audio subject of the Mobile Media API, a new collection of classes dedicated to playing media such as MIDI tunes, MP3 files, and MPEG video clips on mobile devices. The way this API works is through a series of Player objects. You construct a Player by passing a URL for a video file, such as “http://www.flarb.com/blah.mpg,” to the Media Manager object’s createPlayer() method. This function returns a Player object associated with the media type of the URL. This Player object can then control the media, including the position in the file to start playing, looping the stream, and starting and stopping the playback. Whether wireless devices will

Methods' Profiler - Wireless Toolkit

File View

Save Open

Call Graph

- (100.0%) <root>
  - (8.6%) com.sun.midp.midlet.Selector.run
  - (0.29%) com.sun.midp.lcdui.DefaultEventHandler$VMEventHandler.run
  - (70.28%) com.sun.midp.lcdui.DefaultEventHandler$QueuedEventHandler.
  - (1.72%) java.util.TimerThread.run
  - (18.96%) com.sun.midp.main.Main.main
  - (0.11%) java.lang.Class.runCustomCode

Find...

ALL calls under <root>

Name	Count	Cycles	%Cy...	Cycles...	%Cy...
<root>	0	0	0	258138916	100
com.sun.midp.lcdui.DefaultEventHandler$QueuedEventHandler.run	0	2420227	0.9	181423252	70.2
javax.microedition.lcdui.Display.repaint	136	280563	0.1	175682847	68
com.sun.midp.lcdui.DefaultEventHandler.repaintScreenEvent	132	143969	0	174144313	67.4
javax.microedition.lcdui.Display$DisplayManagerImpl.repaint	132	139275	0	174000344	67.4
javax.microedition.lcdui.Display$DisplayAccessor.repaint	132	139139	0	173861069	67.3
javax.microedition.lcdui.Canvas.callPaint	133	754196	0.2	173605282	67.2
c.paint	133	174800	0	171090413	66.2
javax.microedition.lcdui.game.LayerManager.paint	133	1096867	0.4	170915613	66.2
javax.microedition.lcdui.game.TiledLayer.paint	133	1527123	0.5	168662430	65.3
javax.microedition.lcdui.Graphics.drawRegion	34181	167796299	65	167796299	65
com.sun.midp.main.Main.main	0	6185	0	48954705	18.9
com.sun.midp.main.Main.runLocalClass	1	6147	0	46425176	17.9
com.sun.midp.dev.DevMIDletSuiteImpl.create	2	47081	0	41664586	16.1
com.sun.midp.midletsuite.JadProperties.load	4	5532	0	37246869	14.4
com.sun.midp.midletsuite.JadProperties.partialLoad	2	292666	0.1	37239889	14.4
com.sun.midp.security.Permissions.forDomain	2	110452	0	34056793	13.1
com.sun.midp.security.Permissions.getPermissions	1	2427	0	33941334	13.1
com.sun.midp.security.Permissions.readPermissionsTable	1	298467	0.1	33935949	13.1
java.io.InputStreamReader.read	1222	2605146	1	31942169	12.3
com.sun.midp.midletsuite.ManifestProperties.partialLoad	1	1448	0	30011334	11.6
java.io.Reader.read	1222	15895751	6.1	28688421	11.1
com.sun.midp.midletsuite.ManifestProperties.readLine	23	1425109	0.5	27993398	10.8
com.sun.midp.midlet.Selector.run	0	2028	0	22219978	8.6
com.sun.midp.midlet.MIDletState.createMIDlet	2	630	0	22164757	8.5
java.lang.Class.forName	12	20772023	8	20772023	8

The new Profiler gives extensive performance information and analysis of your MIDlet's execution.

have enough memory to store movies and MP3s or the bandwidth to stream them is a whole other issue. Still, it's nice to see some forward-looking support for features that may become more commonplace on handsets in the near future.

Wireless Messaging API

A J2ME standard extension, the Wireless Messaging API, has widespread ramifications for multiplayer game development, as it standardizes several network communications methods that previously were only available in vendor-specific custom extensions and packages. In MIDP 1.0 you were limited to HTTP communication via the Generic Connection Framework. Some handset manufacturers provided lower-level socket access, however this was nonstandard and thus MIDlets using these features had to be catered to each handset's custom SDK. Now, the Wireless Messaging API not only includes HTTP and socket communication, but text-messaging protocols and HTTPS as well. Communication via the serial port is also standardized in MIDP 2.0. This wealth of communication options directly contrasts with the minimal approach of MIDP 1.0.

"Push Architecture" is another one of MIDP 2.0's new features. Although rarely used, a feature of Qualcomm's BREW allows an applet to receive an SMS message broadcast by a remote server. If the applet is not already running, it can either wake up and run, or silently process the message behind the scenes. Otherwise, it receives an SMS message as any other event in the message handler.

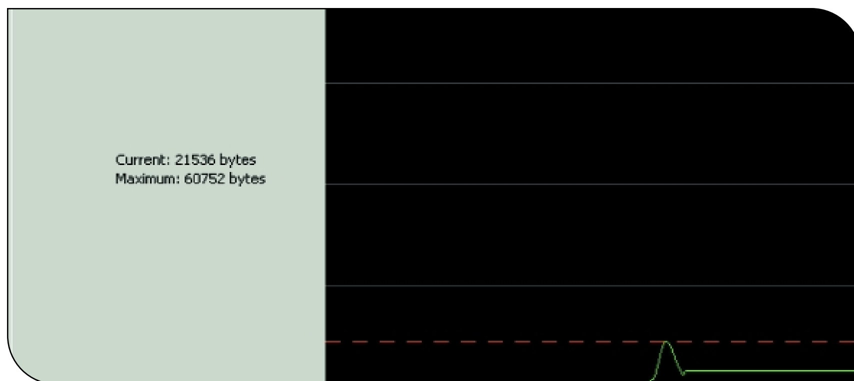
MIDP 2.0 now features the ability to push data to MIDlets, which bears some resemblance to the above-mentioned process. This is done via the `PushRegistry`, a mechanism that allows a MIDlet to register one or more sources that may push data to itself. Inside the descriptor file, you can list a number of different data sources and the class inside the MIDlet suite that they push data to. Then, inside the respective classes, the `PushRegistry` object can be used to create connections to any of these sources as data is pushed to the class. As messages are fed to the applet, they can be read via a `Connection` object and dealt with.

The Toolkit

Along with the advent of MIDP 2.0, Sun has also released an updated version of their Wireless Toolkit. The Toolkit includes the familiar KToolbar IDE and emulator as well as a few new utilities and tools to debug, profile, and digitally sign MIDlets for distribution.

KToolbar is the IDE we all know, but with a few changes. As has been the case with some of the more recent versions, MIDP 2.0's KToolbar has the ability to obfuscate the code before putting it in a JAR. This not only makes it difficult to reverse-engineer the binary, but it crunches down the size of the applet as a side effect. The size savings may be great in some cases or miniscule in others, but saving a few bytes any way you can is welcome in the restrictive world of wireless application development.

KToolbar also has extensive debugging and profiling features as well as emulation performance settings. The Profiler allows you to peek inside the execution of your MIDlet, timing functions, and monitoring memory usage. You can also inspect network traffic via the same interface. The emulator has many different



The memory monitor allows you to examine your MIDlet's memory usage statistics in real time.

performance options that allow you to adjust the execution speed, graphics display latency, and other properties to simulate the behavior of an actual handset.

The biggest problem with Sun's new Toolkit is the emulator's performance. As there are no MIDP 2.0 devices available, I can't judge the accuracy of the emulator, but it runs painfully slow regardless of the performance options selected. My demo of a tile background and a single sprite seemed to run at a miserable two to three frames per second. I refuse to believe next-generation handsets that run MIDP 2.0 will have such horrible performance. Granted, at press time the current emulator was still a beta release. Hopefully, Sun will improve the emulator's performance by the time the final release is upon us.

Provisioning

MIDP 2.0 goes beyond the programming end of things to address the distribution side as well. With MIDP 1.0, every handset manufacturer and carrier was free to implement its own provisioning system. Some manufacturers required the handset to be linked to a PC where the applets were uploaded in a Palm HotSync-style fashion, others had Over The Air, or OTA, distribution of applets. Each OTA system had a unique interface and feature set. Sun has standardized the OTA process providing a mandatory method and interface to the wireless downloading and installing of MIDlets. The OTA interface also allows carrier-side querying of installed applets as well as other features. This is the equivalent of BREW's Mobile Shop — however, unlike BREW, Sun has not addressed the billing and revenue collection issue. It's nice to see some progress being made on this front, but J2ME desperately needs a standardized billing and certification system much like Qualcomm's True BREW process.

MIDP 2.0 vs. BREW 2.0

So, how does MIDP 2.0 compare against Qualcomm's latest version of BREW? They are quite similar in many ways: both contain sprite and tile functionality, both include the ability to transform graphics, both have robust support for various communications protocols; both support rich media playback, and both are only available in emulated phones. We should start to see MIDP 2.0 handsets by the middle of this year, and although I have seen prototype BREW 2.0 devices, we may not see them on these shores until the holiday season. Therefore, it's only possible to compare both from a development perspective, not on real-world performance.

Since both compare favorably to each other in regards to features, it's now a matter of market penetration, the ability to make money with the platform, and your own development preferences that factor. Market penetration is an issue that will play out over 2003 and beyond. Despite J2ME's wider market penetration, Qualcomm's billing system allows you to generate revenue from a much smaller user base. However, if you like Java you will probably be a fan of MIDP 2.0; if you are an old C/C++ hack as I am, BREW may still be preferable. I still like the low-level device access provided by the BREW environment as well as the ability to use C/C++. However, I also admire the ease of use of the J2ME environment and all the convenience the basic language features offer. These features include such things as garbage collection, collection classes, threads, and other standard J2ME features.

We're going to have to wait to see how well the device manufacturers implement both MIDP 2.0 and BREW 2.0 on their new handsets. One thing is for sure, however, the mobile gaming industry is in constant change, and we'll see how these platforms evolve over the rest of the year.

The Big Picture

MIDP 2.0 represents a dramatic step forward in game development for mobile devices. Now it's up to handset manufacturers to adhere to Sun's standard and provide robust and speedy support for MIDP 2.0's enhancements. If the hardware follows through, the days of primitive beep sound effects, single-digit frame rates, bland monochrome graphics, and frustrating controls may be a thing of the past for mainstream Java handsets.

MOBILE GAMING RESOURCES

MIDP 2.0 Resources
- http://java.sun.com/j2me
- www.microjava.com
- www.wirelessdevnet.com

BREW Resources
- www.qualcom.com/brew
- www.loftesness.com/radio/categories/brewlog
- http://developers.verizonwireless.com

General Mobile Developer Resources
- www.wirelessgamingreview.com
- www.midlet-review.com
- www.radio-gamer.com

Effective Middleware Evaluation

My first experience as a middleware user dates back to 1995, when I first used RAD Game Tools' Miles Sound System. Windows 95 hadn't yet replaced DOS as the major gaming platform, and there was no standard sound API. Years have since passed, but many of the inherent challenges of licensing technology remain unchanged. Nowadays, free multiplatform standard APIs are increasingly rare, opening wide the doors to the middleware business. As the game business matures, middleware is addressing more and more of our needs, especially the ones that are more recent to game development: MMO network technology, physics, and advanced AI.

Middleware products are a part of our working environment like any other, but they hold a privileged position. Because they are central to project development, they can represent an important cost. Choosing a technology can be a relatively straightforward task, one which most developers can handle successfully with the right approach. But because choosing a middleware solution is more than just choosing a technology, the evaluation process becomes a delicate exercise, the ramifications more far-reaching.

Be prepared to spend several man-months for a good middleware evaluation. This is the price to pay to minimize risk. The secret of an efficient evaluation is more than simple resource allocation, it lays the strategy that you set up for the task. This article is designed to help developers formulate a strategy that best suits their needs.

Know Your Middleware

Despite the inherent vagueness of the term "middleware," we need to classify its aspects in order to compare different commercial offerings, looking both at how we define the components of middleware and how we distinguish classes of middleware from each other.

Middleware components. The most obvious element of a middleware package is a piece of software, or more precisely a library accessed through an API. Typically, middleware packages also comprise the necessary learning materials in the form of books, samples, and tutorials. Some packages also come with production tools. That is the product part of a middleware package.

In addition to the product itself, there is also the service part. The minimum service middleware provides is technical support, but many providers also offer consulting. Contract work can also be part of the service offer. If you lack the knowledge or manpower to develop one feature of your game, or you don't have time to optimize it yourself, some middleware providers will develop software specifically for you or dispatch a site engineer to spend some time with your team, or offer training sessions.

Middleware categories. The middleware world splits into two main categories: dedicated (to certain kinds of games, such as the Unreal engine) and generic (such as Renderware, Alchemy, Havok, and so on) (Figure 1). The two categories address different needs, so when choosing middleware you have to take care to compare comparable things. I've seen companies comparing middleware offers from both categories in the same evaluation process, clear evidence that the developer's needs were not well defined.

Another common situation is the need to choose between using a proprietary technology and licensing an external one. Let's distinguish two kinds of proprietary technology: the reusable and the recyclable. Reusable technology is developed separate from any project by a dedicated team. Its goal is to follow the scheme of commercial middleware but is restricted to internal needs.

Recyclable technology is, most of the time, a game engine designed for a specific game with poor reusability potential (poor documentation is a common culprit). Nevertheless, cost savings tempt developers to consider recycling it. From a strategic standpoint it doesn't make sense to compare such a game engine with commercial middleware.

Frequently, developers also consider a third kind of proprietary technology: the imaginary one. It's the perfect technology that some developers dream of programming. But as far as I'm concerned, I

ALEX MACRIS | *Alex has seen many sides of the young middleware business. He debugged his first game program in 1994 at Cryo. There, he spent many years co-engineering Cryogen, Cryo's in-house middleware used in more than 20 games. In 2001 he joined Intrinsic Graphics' developer support team, where he assisted many studios in their middleware evaluation. Alex now works as an independent middleware consultant. Contact him at alex.macris@wanadoo.fr.*

don't know how to compare middleware with dreamware.

Know Your Evaluation Needs

Evaluating middleware offers is not necessarily very complex, but it still needs a lot of attention. Developers should give particular care to the evaluation environment and resources.

Evaluation's actors. There are three parties involved: the good, the bad, and the ugly. At first sight we could think that there are only two: the middleware company and the game company. But most of the time game companies' interests or publishers' interests may diverge from the game project's constraints. Thus two parties may represent each side for the studio: the producer/publisher/technical director on one side, and the project manager on the other. This point is very important in terms of negotiation diplomacy; middleware offers are evaluated twice.

Technical directors have a global approach. They do not think on a single-project basis but on a multiple-project one. For example, let's imagine that a new and very promising middleware product has come on to the market. There is a game company that uses a well-established middleware solution for all its projects that it finds largely satisfactory. This company's technical director may decide that it is strategically important to test this new middleware, but he wants to do only one in vivo test, because it is risky for the chosen project. The project manager on the other hand has the mission to succeed in developing a game, and probably doesn't want to see his or her project chosen as a guinea pig. Keep this in mind: project managers have to anticipate the fact that they may have to live with middleware which is not the solution they prefer, so the scope of their evaluation changes.

Thus the project manager doesn't simply need to find out which middleware responds best to the project's needs, but rather how far away each middleware solution is from his or her needs. With this in mind, when the choice is eventually made, the project manager immediately knows where to direct the team efforts to fill the gaps that the chosen middleware solution leaves open.

Evaluation reasons. Making an effective decision also depends on the reasons your studio is licensing middleware. For example, one rationale for licensing technology that rarely gives good results is when there is a recognized gap between the ambition of a project and the experience and technical talent available on the team. Some think that licensing technology will fill this gap, but the truth is you cannot realize a great game without talented people, and middleware products need talented people to leverage their power.

Technologies / Game Families	3D graphics	Sound	Physics	AI	Network	Other
RTS						
FPS						
Adventure						
Racing						
Sports						
Etc.						

Dedicated Middleware — Generic Middleware

FIGURE 1. Dedicated middleware provides all the technology needed for a unique game genre. Generic middleware provides a unique technology that is designed to fit the needs of any kind of game.

The most common reason cited for licensing an external technology is decreasing game development costs and saving time. This is not always the outcome of licensing middleware. Don't hope to save a lot of money on the first shot. Taming an external technology takes a lot of time and a lot of energy. This is mostly true for generic middleware. Dedicated middleware, which is closer to the final game software, benefits from a shorter trial period.

Another reason you might be licensing middleware is the specialization of tasks. Today we clearly distinguish the professions of game development and technology development. While one must necessarily know about the work of the other, knowing is not doing. Middleware discharges the game developers from the task of creating technology, letting them focus on gameplay. During past evaluations, I have faced situations where the key point for the customer was the capacity of the product to receive new pieces of technology that were developed by the game team. Just as game programmers long ago gave up creating their own art, so too should they learn to distinguish between technology development and game development.

Evaluation resources. Let's try to calculate the amount of time needed to evaluate just a single product. At the very least, you have to test the technology, the API, the tools, the documentation, the performance, and the feature set, and you have to do this work for each platform. My experience estimates the time needed to cover these bases adequately at 20 to 40 working days. Then, you have to do that for each competing product. If there are three competing products, you end up with a potential six-man-month evaluation time.

Besides time, the other resource consideration is manpower. If you can only allocate a single person for this work, assume that between the beginning and the end of the evaluation, the products will have evolved. Then, if you want to license two generic pieces of middleware of different kinds (such as graphics and physics), you have to evaluate their capacity to cooperate and work together. This process takes even more time than you can imagine.

You also need to test the service, which means setting up a list of test questions that you will submit to each middleware provider's developer support service. Distribute your questions all

along the evaluation period. Waiting until the end of the product evaluation to post questions (a common habit) means lengthening the overall evaluation time.

It's critical not to underestimate the resources needed for the evaluation; this is the single biggest cause of poor decisions.

Determining the Scope of Your Evaluation

The amount of resources you can assign to the evaluation exercise is naturally limited. Let's see how you can utilize resources to get the most economical and informative evaluation possible.

Not one, two evaluations. You should remember you are evaluating both a product and a service. Most of the time, developers only evaluate the product; evaluating a service is difficult and not always relevant. Service is a cost to middleware companies, so some may be reluctant to spend too much effort with evaluation processes. At the other extreme, other companies seem to expend more effort supporting potential clients than in supporting their actual customers.

By taking the time to evaluate the service, you can't be blamed for choosing a middleware provider that has overestimated its capacity to provide services. In short, a game company should choose middleware based on both evaluations, but should keep it for the next production because of their satisfaction with the service.

Not two, three evaluations. Besides evaluating both the middleware product and service, you must also do a thorough and honest evaluation of your needs. This step is essential but so often overlooked. This aspect of the evaluation is required, because there is no good or bad middleware — only middleware that addresses problems and then aims to solve them.

All three evaluations (your needs, the products, and the services) may be done together, and in fact middleware companies can help you in evaluating your needs. Involving middleware companies' developer relations staff in identifying needs is a good way to evaluate the expertise part of the service they offer. Middleware providers may understandably be tempted to bias their advice to influence your choice. But remember that it's not in the technology provider's interest to disguise your real needs too much. In a small industry where the fame of any product is fragile, they don't want to risk turning you into an unsatisfied client.

Evaluating your needs should answer a fundamental question: Are you making a long-term or a short-term choice? For a short-term choice you want to focus on the robustness of the technology and on its capital, not on its potential. For a long-term choice you need to distribute the risks across a longer period, thus making the extensibility of the product a key point. Extensibility can be measured through both the product (its technology, the way it is architected, the roadmap, and so on) and the service (the size and the talent of the engineering team and its capacity to respond to both the customer needs as well as the hardware evolution).

Determining short-term versus long-term needs will also answer the question of to what degree you will need to stay independent of your middleware solution. How important is it for you to be able to step back and change your middleware choice after a year or two? If you need custom tools, where should they be developed, on the game company side or on the middleware provider side? Any tools developed by your middleware provider would need to be redeveloped if you changed middleware. A work supply is good, but it's also a way for middleware companies to increase their customers' loyalty. On the other hand, keeping your tools independent of an external technology has a cost too.

Product Evaluation Checklist

Following is a list of the main points to hit during a middleware evaluation. It doesn't take into account one of the biggest points, cost, but it is designed to help you figure out exactly what and how much you are getting for your money.

Evaluating performance. Performance has been at the heart of the game industry since its earliest days. Thus it has long been a passionate subject for developers. Today, however, developers may be overestimating the importance of performance at the expense of the importance of the production tool chain. The main thing to evaluate with middleware performance is what you achieve with the middleware, not what the middleware achieves. It is a very common mistake to compare peak middleware performances. Performance optimization strongly depends on knowledge and time spent on it, so what you want to find out is what performance you can achieve spending a known amount of time.

Evaluating performance in relation to time spent with middleware is difficult, but the main thing you want to find out is how easy it is to leverage the power of the middleware. The only way you can do this is to give a try. Set up a benchmark and test it over the middleware candidates, then try to optimize it for each one and see how easy it is. Get help from the middleware support teams, and be persistent with your own work. Many evaluators are tempted to pass the bench protocol to the middleware support team, asking them, "What performance can you

ESSENTIAL MIDDLEWARE WISDOM

- There are no inherently good or bad middleware solutions, only offers that fit or don't fit your needs.
- Choosing middleware is a difficult, long, expensive, and critical job. Do not underestimate it.
- Licensing middleware is buying both a piece of software and a service. Don't underestimate the service, and more than that don't forget to evaluate this service.
- Avoid some classic traps such as performance misevaluation and extensibility misevaluation.
- Project managers should be prepared to work with middleware they haven't chosen. Evaluating doesn't always mean choosing.

achieve with that?" Thus optimizations are achieved by very experienced engineers who know the product very well.

Such results reflect only the peak performance of the middleware, which does not inform you about the speed of the technology once your own team has implemented it in your game. Figure 2 illustrates how your real-world implementation can lead you toward a different decision from simply favoring middleware with higher peak performance. Given the capabilities of your team, it is the difference between analyzing average performance versus peak performance, and analyzing the capability of the product to be leveraged by your team.

Evaluating the learning curve. The preceding example illustrates the importance of evaluating the learning curve. This point is even more important if you are making a short-term choice, where anything that helps shrink the learning curve is welcome. Again your team's unique experience comes into play: don't underestimate the significance of having some programmers who have prior experience working with a certain kind of middleware.

The ability of the middleware product to conform to standards is also an important factor. For example, does the middleware product use STL, or does it use some custom container sets? Many programmers already know how to use STL and thus shouldn't have to waste time learning a new container API.

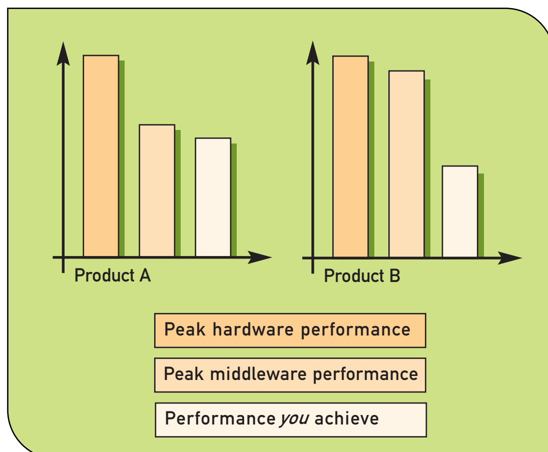


FIGURE 2. Theoretical results of two middleware comparisons.

Evaluating the technology. Another point of interest is the technology on which the middleware product is based. This is a passionate issue. There are several philosophical schools out there (C versus C++, consoles versus PC, and so on), and it's not common to betray your alma mater. But this is another case of the game industry changing faster than many developers' minds. An evaluation must keep in mind that technology is only one aspect of an overall middleware solution. Don't neglect a product because its technology doesn't belong to your philosophical school of thought.

Evaluating extensibility. Conventional wisdom suggests that extensibility, potentiability, replaceability, dreamability, or whatever you want to call it, is a good quality for middleware. This thinking is so common that I've assisted in evaluation discussions (from both the developer's and the provider's side) where the desire for extensibility expressed by the customers ended up as: "We want to be sure that we are allowed to extend or replace every piece of your middleware." A translation of this question could be: "I want to buy a $100,000 product from you that I have the desire to rewrite entirely."

Extensibility is only good if it's needed; it's bad if it's not needed, because building something extensible means integrating as many constraints as possible, leading to more and more compromises. Extending middleware is not necessarily a goal unto itself, so it should be avoided as much as possible. Developers must keep in mind that extensibility is just one of many aspects of a middleware product to be weighed.

Using a product and extending it do not have much in common. Most of the time, extension mechanisms are poorly documented, poorly tested, and poorly respected (generally, be prepared to rework all your extensions at each new release of the product).

Your middleware solution also may need a very high level of expertise to extend. Based on my range of experience, my advice is to keep as far away as possible from the temptation to extend a middleware product. This work is better done by your middleware company as part of your work supply contract.

Source code or not? The question of whether to obtain source code is always a sensible one. Most of the time middleware companies are reluctant to provide it, but many customers express the desire to have access to it. Each side has its obvious and nonobvious rationales behind the source-code question.

The game developer's main argument is that source code addresses the need of easy debugging. This is a genuine need, and source code indeed helps, but there may be better ways to solve this problem. Another argument is that source code helps the developer understand how to use some features or how to extend them. More than arguments, source code may provide a psychological comfort for those whom black boxes frighten.

On the other hand, middleware companies also have their own sets of arguments against supplying source code. First, they don't want to support a modified product. Also, they don't want customers to make assumptions about how features work. They want to be free to change any piece of code (keeping the functionality and the interface intact) without breaking something in the customer's realization.

Thinking more deeply about the utility of having access to source code reveals more unexpected cons than pros. The main con is time. When customers are tempted to seek an answer to a question directly in the context of source code files, they must face potentially hundreds of such files that the average middleware consists of. Compare that time spent with asking a question to the provider's developer support team, who should be more familiar with the middleware source code. Seeking information or bugs in the middleware's source code is not the game developer's job; it's the middleware provider support team's job — development customers pay for it.

There is one situation where you absolutely need source code, which is when the middleware company goes bankrupt. Be sure that your contract gives you access to the source code if such a situation occurs.

The goodies. The last point on the product evaluation checklist concerns the goodies. Not T-shirts or key rings, but more serious extras. All developers have a pet aversion to some parts of the development of a game. Localization is an example, or perhaps big files or memory card support. Even having access to a rich mathematics library is a nice extra. Some of these functionalities are offered by middleware products. Depending on your team, they can add up to the difference between a good choice and a very good choice.

Don't Forget the Service

For all the focus on technology in the game business, service is the heart of middleware business, and it's also the Achilles' heel of many middleware offers. Don't give short shrift to evaluating the service aspect.

Evaluating the developer support service. The simplest aspect of developer support you must evaluate covers such bland communication as, "Hello, there is a bug in your SDK." Or perhaps, "I'm testing the client-server feature but it doesn't work." "Have you plugged in the network cable?" "Oh! No! Thanks." You will probably stress this service a lot in the beginning of the evaluation, as it is the period when you have many questions to ask. The ratio of customers to support engineers can also help you to evaluate a provider's potential availability.

But beyond those basic questions, think of a developer support service as a critical resource, and don't procrastinate until October or November to ask questions — you won't be the only customers facing a holiday crunch. During non-peak times, a fair developer support team should be able to respond to you in an hour, at least to tell you that they are handling your request. But developer support alone can't handle all types of requests. Its limit is when you ask, "How do I do that?" Developer support's job is in part to give you an answer, but it is also their job to pull you to another level of service: the expertise.

Evaluating expertise and consulting. The expertise service is there to orient customers. At this level a typical answer to the customer's question "How do I do that?" is "Why do you need to do that?" Developer support helps you implement a solution, the expertise level helps you find the solution. A middleware company is a partner that can help you in analyzing your needs. Ask a potential provider "How do I do that?" and if you don't get the right answer, the middleware company might not be conscious enough of their role.

By understanding your needs, your middleware partner can point you to a different solution; they should know that their product will respond better to one method over another. They may have already faced the same problem with another customer. Used intelligently, such services can be of a great help, efficiently complementing an internal R&D team, as the middleware company's sources of experience are potentially more diverse.

Consulting is a step beyond the expertise level, able to address problems not related to the product, such as: "Help, it's my first game on PS2 and I have no experience setting up an MMP platform!" In the middleware provider's role as consultants, you may also be able to benefit from their network of acquaintances.

Evaluating work supply. Last but not least is the work supply evaluation. This is a key point if you need to extend a middleware product significantly. It can also be very useful if you need some extra resources to finish your game on time. In this case you need immediately operational staff, which makes it hard to evaluate far in advance, but you can take some precautions. At least try to anticipate your future needs and specify further work into the contract. List all the extensions, features, and tools you will need, and list all the project's phases where you will need an expert on your site. For example, if you are already sure that a short period before your alpha you will need to do an optimization pass on your game, schedule that into the contract.

Final Advice

Beyond process-specific evaluation strategies, there is also some overarching advice to go with any company evaluating any technology partner. First, go on-site and visit the middleware company's headquarters. Visit the engineering teams. Talk with them.

Abuse the intimate nature of the industry. Pick up the phone, call friends working for other companies, and ask them what their experiences are with a given middleware solution.

Ask middleware companies about the level of renewal of licenses. If you ask them for their number of customers, they may answer you with a number that may seem impressive but really lacks context. Instead ask them how many satisfied customers they have who have renewed their licenses.

Think of the ramifications on your hiring and training strategies. If you can easily find game developers that already have experience with a given middleware product, that's a point in its favor.

Above all, remember that the middleware market is still at a very early stage. It changes very quickly, so keep evaluating middleware on a regular basis.

FOR MORE INFORMATION

Generic middleware list (incomplete but thorough):
www.middlewarenet.com

Specific middleware:
www.idsoftware.com
www.unreal.com

All-in-one middleware:
www.virtools.com

ACKNOWLEDGEMENTS

Thanks to Stuart, Laure, Cecile, Ben, Olivier, Manu, and Toni for their help reviewing this article.

The Character Development of Sucker Punch's SLY COOPER AND THE THIEVIUS RACCOONUS

GAME DATA

PUBLISHER: Sony Computer Entertainment
NUMBER OF FULL-TIME DEVELOPERS: between 14 and 24
NUMBER OF CONTRACTORS: up to 4
LENGTH OF DEVELOPMENT: 3 years
RELEASE DATE: September 2002 (U.S.), January 2003 (Europe), February 2003 (Korea), March 2003 (Japan)
TARGET PLATFORM: Playstation 2
DEVELOPMENT HARDWARE: 1800+ MHz PCs with 512MB RAM, 40GB hard drives, and Nvidia Quadro 4 graphics cards
DEVELOPMENT SOFTWARE USED: SN Systems' ProDG, Maya, Photoshop, Visual C++ 6.0, Visual SourceSafe
PROJECT SIZE: 3,000 Maya files (1.5GB); 4,300 textures (800MB); 1,000 source files (9MB)

After the somewhat-successful 1999 holiday release of ROCKET: ROBOT ON WHEELS, Sucker Punch's inaugural title, we were eager to start on our next game. We'd settled on doing another platform adventure but knew we'd need a more compelling lead character to compete with the great titles sure to come to the Playstation 2.

We expected that defining a great character would be a challenge for our team. We would be doing lots of things for the first time: building a new PS2 engine, switching to Maya as our authoring environment, casting voice actors and doing lip sync, animating a fully articulated character, and designing for a worldwide release. It turned out to be more of a challenge than we ever anticipated. Without three years of hard work from everyone at Sucker Punch, and without reams of useful feedback and support from our production teams at Sony, we would have been lost.

This Postmortem focuses on the genesis and development of our lead character, Sly Cooper, a raccoon thief from a long line of raccoon thieves. We hope that focusing on a single aspect of the development of SLY COOPER AND THE THIEVIUS RACCOONUS will provide an interesting change of pace from the more general project-wide Postmortems that usually run in this space.

What Went Right

1. Using interaction with other characters to define Sly. In early iterations of the game, Sly wasn't a very convincing character. We'd given him thiefy abilities — breaking into safes, swinging from hooks, dodging security systems — but it wasn't enough. Sly felt more like a puppet than a character. It was fun to drive him around, but we were well short of the immersion we sought after.

The breakthrough came when we started using Sly's interactions with other characters in the game to define Sly to the player. It's difficult to define a character in isolation; it's much easier to define a character by contrasting him with other characters, both allies and adversaries.

Sly isn't acting alone in his criminal escapades, he has two partners, Murray and Bentley. Bentley the Turtle is the brains of the outfit; he's very cautious, and — there's no way around it — a complete nerd. In contrast, Sly is smooth and a little reckless. Murray the giant pink hippo is the innocent burden of the team; he's bumbling and clumsy, where Sly is lithe and agile.

Sly's adversaries perform a different function — they help maintain the ethical tension of being a thief. We liked the fuzzy morality of Sly. He's not exactly a hero, but he's not exactly a villain either. We use the Inspector Carmelita Montoya Fox character to make sure Sly doesn't feel like just another good guy. Her primary function is to jump out and yell "Criminal!" every so often, just to remind players that they are, in fact, breaking the law. Otherwise, it's easy to forget Sly's a thief. Worrying about the ethical basis for the character's actions isn't something platform gamers do very often.

On the other side of the spectrum are the boss characters Sly confronts. The bosses are ruthless, rapacious, and over-the-top in their disregard for all decent standards of propriety. In comparison, Sly doesn't seem so bad. Sure, Sly's stealing stuff, but at least he isn't burying an entire village under an avalanche, like the Panda King character does.

With Carmelita on the good side and the bosses on the bad side, Sly becomes a more interesting character, not purely good or bad, but with elements of both.

2. A seemingly simple control scheme. A combination of factors pushed us toward a simple control scheme. First, we wanted the game to be accessible to both young and inexperienced players. Second, we wanted the game to be fun rather than challenging, and trying to remember complicated control mechanisms didn't seem fun. We settled on a three-button scheme: X to jump, Square to attack with the cane, and Circle to trigger a thief move.

The thief moves were the tricky part. The Circle button is contextual; pressing it has different effects depending on where Sly is in the environment. Pressing Circle while jumping near a rope causes Sly to grab the rope; pressing Circle while standing next to a chimney causes Sly to crouch and hide behind the chimney.

Guessing what players intended when they pressed Circle was crucial. If we guessed right and Sly did what the player intended, then it wouldn't be very different from the character jumping when the player pressed X. If we guessed wrong, everything would fall apart. The player's self-identification with the character would immediately be broken, and Sly would be just a balky puppet.

We attacked this nasty relationship between action and expectation from both ends. First, we visually marked areas where Sly could do something thiefy with a distinctive blue particle emitter. Obviously, this helped prompt the player through the game, but on a more subtle level it disguised all the places where thiefy actions were not allowed. We didn't allow the player to crouch behind all objects, just in places where crouching was interesting. This vastly reduced the number of possible guesses when the player pressed Circle, and made it much easier to guess the player's intention correctly.

At the other end of the problem, we ran into situations where more than one thiefy action was plausible. We might have two pipes running fairly close to each other, with the player jumping between them and pressing Circle. Which pipe should Sly grab?

If Sly's standing on the ground, choosing the closest pipe works pretty well. If Sly's in midair, however, the answer isn't as obvious. Sly might be closer to one pipe but moving away from it toward the other pipe, which happens to be a couple of meters below. Divining the player's intention in cases like this is problematic.

The obvious solutions, such as always choosing the closest object, weren't foolproof enough. Our eventual solution was to choose the thiefy alternative that required the least amount of air steering. This matched player expectations pretty

CHRIS ZIMMERMAN | *Chris is the development director at Sucker Punch. Sad to say, of all the characters in Sly Cooper and the Thievius Raccoonus, Chris most resembles Bentley. Contact chris_zimmerman@suckerpunch.com.*

well, as it involved the smallest application of physically implausible air steering. So, Sly would be likely to choose the lower pipe in the between-two-pipes example, since the player is already headed toward it and there's plenty of time to make any small course adjustments as the player falls to it.

3. Making Sly look great.

Our art direction for levels was consistent throughout the development cycle. We wanted dense, graphically rich worlds for Sly to run around in. The backgrounds would have lots of color, lots of movement, and lots of shape. Early in development, we did two full-color conceptual paintings, one of an exterior environment and one of an interior, which acted as touchstones for the graphical look of the game. From then on, we were just trying to make a game that looked like those paintings brought to life.

The danger of all this background richness was that Sly might get lost in it. We couldn't afford to have the environments look so good that Sly looked weak in comparison. In short, Sly needed to be the most attention-grabbing and attractive thing on-screen. Three techniques of the dozens we tried were notably effective.

Cel borders. Drawing cel borders around the characters (actually just a dark gray outline) really drew the eye to them and separated them from the background.

Color and texture. We used color and texture to distinguish Sly from the background. The Sly model uses simple, relatively low-detail textures. The dominant colors are cool blues and unsaturated grays, with bright red and yellow accents. Backgrounds, on the other hand, use much higher-detail textures. Colors are much warmer and more saturated. We found these differences to be subtle but effective.

Tail. Given the way our camera system works, a player spends most of the time looking at Sly's rear end, so the tail was going to be a focal point for most of the game. We wanted the tail to be big and bushy, but having it as a big part of the visual look of the character made it crucial to get the tail's movement right.

Initially, we expected to hand-animate the tail, so we spent an unfortunate amount of time writing a spline controller plug-in for Maya, with accompanying support in our tool chain and run-time engine. The results were disappointing — it just didn't look like a tail, despite our animators' best efforts. In fact, it looked like someone had stapled a big, striped sausage to Sly's butt. This wasn't exactly the look we were going for.

Next, we tried simulating the tail, which was much more successful. Basically, we treated the tail as a chain of particles in world space, with distance constraints between adjacent particles, velocity damping in world and local space, and spring functions that try to straighten the tail. In most of Sly's animations, we animated the tail by wiggling the first tail segment; the rest of the tail wagging along falls out of the simulator.

The results were emotionally satisfying. Not only did Sly's tail move much more gracefully in each animation, it also looked great during transitions between animations, when Sly jumps up and down, when running in circles — whatever Sly did, the tail reacted in a believable way.

Sly sending a guard to his E-rated demise.

When it comes to sneaking around, nothing beats ... a barrel.

4. A rich animation toolkit.

We used a simple skin-and-bones system for Sly's character model, so his movement boiled down to transform animation on the bones. It all seemed so simple to us at the start of the project, but we managed to find ways to make it complicated by the time we were done.

Smoothing. We didn't want to have Sly pop between animations, but we also needed him to react instantly to player input. We ended up adding smoothing to our animation controllers. Our smoothed controllers don't lock the bone to the animated keyframes; instead, they smoothly move the bone toward the animation. Once the bone gets to the animation, it locks onto it, and thereafter plays the animation as originally authored.

Blending. One of the drawbacks of hand animation is that you can only afford to do a small number of animations, which have to be applied to a huge number of situations. We leveraged animation work by blending between animations. For instance, we continuously blend between a walk cycle and run cycle at partial joystick deflections.

Layering. In some circumstances, we needed to play two animations at once. If Sly whacks at something with his cane while running, we need to play the whack animation while still playing the run animation. We have a simple priority system to handle this. For example, both animations have keyframes for the right hand, but the whack animation runs at a higher priority, so it ends up controlling the right hand.

Programmatic control. We ended up programmatically controlling lots of bones, especially when Sly is standing still. We move and rotate the feet to keep them on the ground, for example, and we shift Sly's body and hands around to make it look like he's balancing when he stands on a moving object.

In the end, all of these features were

mixed and matched in arbitrary ways. For instance, when Sly is grinding down a slippery vine, we peek ahead down the vine to see how much Sly will need to accelerate as he goes around corners. We used this to decide how much to blend between three animations — an animation of Sly leaning right while grinding, one of him leaning left while grinding, and one of him grinding straight. So Sly anticipates and leans into turns, just like players would expect.

Players never notice all of this elaborate technology; they just see Sly swooping gracefully around corners. The goal of all the refinements we put into Sly's movements was to make the technology invisible, so that Sly's personality could shine through.

Not trusting players to think for themselves, Bentley goes step by step through his complex plan.

5. Doing pencil animations first. In the course of building the 230 or so animations that comprise Sly's move set, we built some pretty ugly animations. I was personally responsible for most of them — well, maybe all of them — when I needed placeholder animations to match new Sly move logic under development. (Readers who can identify which Sly animations in the game are actually "placeholders" get extra credit.)

In general, we found it difficult to conceptualize the animation while working in the animation tool. Even after discussions between the programmer, animator, and 2D artist working on the move, we had a hard time doing animations that met everyone's expectations. We found that having some sort of sketch or visual outline of Sly's movement invariably produced more dynamic results.

Eventually, we settled on doing pencil animations for all of Sly's moves. Typically, this meant sketching the extreme poses during the move. So, for a simple attack animation, the extreme poses were the furthest extent of the wind-up, a pose at contact with the target of the attack, and the fullest follow-through pose.

Doing an animation workflow based on these poses was much easier. The extreme poses were a much higher bandwidth means of communicating the feel of the move between the people working on it than words ever were. Timing needed to be set and animation between the extreme poses needed to be defined, but the first cut at animation was usually pretty close to final quality. A couple of rounds of refinement usually got us to a stylish and polished animation that satisfied everyone.

An unexpected bonus was that these pencil animations were extremely useful to product marketing in the end game. We used the pencil sketches as source material when outsourcing work on magazine covers, manual art, and other marketing materials.

What Went Wrong

1. Too much complexity. Not surprisingly, the Sly model is the most complicated one we built. A typical NPC model in Sly Cooper has six to 10 animations; the Sly model has about 230 separate animations, with roughly 10 to 15 total minutes of animation. Tools and techniques that worked well on simple files broke down with the Sly model.

In addition, the interface between the Sly code and the Sly model is much more complicated than anything else in the game. The code makes assumptions about how the model is constructed, how the animations are named and built, and so on. Breaking any of these assumptions caused Sly to stop functioning, sometimes in mysterious ways.

Finally, there were lots of people involved with any change to the Sly model. Typically, this meant a conceptual artist, an animator, a coder, and usually the game designer who'd be incorporating the new move or capability into a level. That's a lot of cooks in the kitchen. This technical and organizational complexity made it harder to work on the Sly model than on any other model in the game, which made progress painfully slow at times.

Some of this complexity seems unavoidable, even in hindsight. In many cases, though, we built in more complexity than was necessary. For instance, our initial implementation of Sly's walk cycle derived his target forward velocity from the velocity of the ball of the left foot at the keyframe where it first touches the ground, with the somewhat misguided goal of reducing foot-slip while walking. Needless to say, the animation team had a hard time keeping track of rules like this, especially since we had slightly different (but just as complicated) rules for Sly's target velocity during other animations.

Eventually, we went to a simpler model. We animated Sly moving forward during the walk cycle, then stripped out that animation channel, which then defined Sly's target velocity at run time.

2. Run-time-only features. We put a lot of effort into giving Sly a distinctive look and feel in our run-time engine, and we feel we succeeded. However, there was one huge drawback: none of this custom code ran in our authoring environment. All the custom look, all the custom behavior, was run-time only. In order to see the effect of a change to the Sly model, we needed to compile it into a level, then load the

level into our run-time engine.

This made texturing Sly and the other models in the game tricky. The colors in a texture change radically when lit by our shading model. Running Photoshop on a PC you see one color; running through a level with lots of blue lighting and purple fog on an NTSC TV, you see a completely different color. The only way to test a texture change was to compile and load a world with representative lighting and see what Sly looked like.

Refining Sly animations was also frustrating. In order to see how an animation really worked, we needed to compile it and try it out in a level. For specialized moves, we needed to compile a level where the move applied — it's hard to test a new pipe-climbing animation without a pipe to climb.

Level compiles were relatively fast — from 15 seconds up to a couple of minutes — but when multiplied by the number of times we needed to preview a model change over the course of a day, the time really added up.

3. Not enough expression. We used morphing to do facial expressions and lip synching for our talking head sections, which turned out to be barely adequate. Managing the morph targets was painful, we were limited to a relatively small set of expressions, adding new expressions was difficult, and performance was sluggish — in general, life was not good.

Given how clunky our technology was, we were happy with the results we managed to obtain. However, the performance problems and general flimsiness prevented us from having any facial expressions during gameplay, which hurt the game. During gameplay, Sly's head is completely rigid (except for his ears, where we hacked in morph targets).

The worst manifestation of this problem is Sly's eyes. They don't move, the face doesn't move around them, and we don't do any special shading on them. The net effect is a completely blank, raccoon-in-the-headlights look. We may have a pretty convincing body, but Sly's head looks like a marionette's. Luckily, he spends most of the game facing away from the camera, so this problem isn't as bad as it might have been.

Carmelita Fox — Sly Cooper's token nemesis/love interest.

4. Performance, or lack thereof. All the depth and complexity in Sly's animations and behavior severely taxed our run-time engine. Sly has roughly 45 bones, which is twice as many as most of our other characters have. He also has lots of vertices to relight, and lots of custom behavior. The net effect was persistent performance problems in every level Sly appeared in. Unfortunately, that describes pretty much every level of the game.

In general, we spent roughly 15 to 20 percent of every frame just dealing with Sly. Taking this big of a chunk out of the performance budget limited what we could do in the levels.

5. Late addition of moves. Most of Sly's moves were in place early in the development cycle. We were good about not changing the functionality of basic moves, although we tweaked the visuals relentlessly for the whole project. Once we had levels laid out, we couldn't change things like jump height or length without breaking things.

There was one huge exception to this. We scattered "clues" through most of the levels in SLY COOPER. Players who find and break the bottles containing all of the clues are given the combination to a vault. Inside the vault is a page torn from the *Thievius Raccoonus* (the stolen family heirloom Sly is trying to retrieve in the game), which grants the player a new power-up move.

We went through most of the development of the game without worrying about what prize the player would get for collecting all the clues, so we had most of the game levels built by the time we started working on the power-ups. That made designing the new power-ups extremely constrained. The power-ups needed to be fun, but they couldn't break any of the game design of existing levels. Basically, they had to be fun, but not very powerful.

In the end, we were able to invent 25 power-ups that met the constraints, but the results weren't well integrated into the rest of the game design. Instead, they feel like afterthoughts.

Stealing Victory

If we need a good laugh, we can always dig up an old build of SLY COOPER, say from about the spring of 2001, and start playing the game. It's hard to not laugh at ourselves, because we thought that Sly was pretty cool at that point. With the perspective of another 18 months of work, we can see just how pathetic he really was — especially if we choose one of the sausage-tail builds.

It would have been nice if we'd nailed the character right away, but that's not the way things worked. We worked on getting Sly right from the day we started the project to the day we went gold. Sometimes this meant taking a feature that seemed perfectly acceptable and trying to make it better still, or jettisoning some move that we'd all gotten attached to. It took a team-wide commitment to end up where we did, and we were gratified by being awarded the Game Developers Choice Award for "Original Game Character of the Year" at this year's Game Developers Conference.

Game Mobility Needs Code Portability

The advent of mobile devices has changed many rules of traditional game development. Several of the stigmas associated with the industry are being broken, and new opportunities arise to make games based on creative decisions rather than purely commercial ones. The technology being reminiscent of game development during the 8- and 16-bit eras in many ways, mobile game development poses significant challenges to everyone involved. Such challenges have attracted many veteran game developers to the mobile development, as they are most familiar with many of the demands imposed on developers.

One such demand is the ability to write portable code. Many of today's game programmers still write their games essentially with one platform in mind, and format conversions end up a sore afterthought. In the mobile space, however, developers realize two things very quickly. First, there are countless platforms out there, and since they are generally not compatible in any way, each requires a tailored implementation. Every handset requires a custom build with dedicated art assets. The variety of OSes on these devices, ranging from BREW and Symbian all the way to Windows CE, to name but a few, also adds flavor to a mobile developer's life. The second thing developers realize is that if they want to be profitable they will have to support a number of these platforms. Given all the idiosyncrasies of each platform and device, this can expand into quite a challenge.

There are different approaches to this situation. One of them is to use a middleware package that abstracts the platform-dependent layer of the code from your game implementation. Though generally a very good way to start things, such technologies may not be suitable for all cases. Fathammer's X-Forge engine is a great example of how developers can obtain a top-tier 3D engine that runs on Symbian phones, Smartphones, Pocket PCs, Palm OS 5 devices, and mobile Linux handhelds. Using such an engine, you are covering a lot of ground already with a minimum of work.

Another possibility is Mophun, a game API developed by the Swedish company Synergix. While it is currently available only for the Sony Ericsson T300 handset, this engine will soon be implemented on a variety of handsets, giving game developers greater opportunities to write portable games.

continued on page 55

Illustration by Dave Whamond

continued from page 56

But no matter whether you hard-code every line or use some of the APIs and middleware available, if you never gave much thought to writing portable code before, now is the time. Here are some thoughts to get you started in the right direction.

Create sensible coding guidelines. Very few game programmers adhere to any sort of coding standards, and very few companies have traditionally enforced them. Start now. Create coding standards for your company and make sure they are sensible. Get rid of all the bad habits and start writing "clean" code. It sounds simple, but you will be doing yourself a big favor. Having proper coding guidelines in place is the first step toward writing portable code because the most valuable coding standards automatically enforce practices that intrinsically make code much more portable.

Never try to `#ifdef` your way through your machine-dependent code. This is a favorite bad habit that has been taught throughout the years, but it is in fact tedious and error prone. The last thing you want to do is mess with a fully functional and tested implementation for one platform, simply because you're interleaving a new implementation for another one in the same source file. Better to separate each device implementation into separate files that are then included at compile-time using the compiler preprocessor.

Everyone who has ever written games on the Apple II, the C64, the Atari ST, the Amiga, and the IBM PC simultaneously understands how important portable code is — and how to design and write it.

Start using constants. Much of the code I see every day is practically hard-coded to specific device specifications of the desired target. It's time you started using constants — and I am talking about real constants, not the C/C++-abomination `#define`. Use your constants liberally instead of using hard-coded coordinates, dimensions, sizes, and so on. In your game, use these constants and derive all other values from those constants if needed. If you separate these constants from the rest of your game code, then suddenly, by editing a single source file, you can create an entirely different version of the game in the blink of an eye.

Create an asset and workflow structure. While most projects have a directory structure that has been established at one point by the developer, few of those projects are thorough enough to accommodate different versions of the same assets. Make sure to create multiple directories for your final art assets — one for each platform. Create subdirectories for your project and make files — for each platform. Create source subdirectories for your machine-dependent code — for each platform.

With these points in mind, you are already better prepared for cross-platform development than most developers in the industry, and it will pay off. It's easy to understand why so many veteran game developers are flocking to develop for the mobile market. Nostalgia may be one reason, but everyone who has ever written games simultaneously for the Apple II, the C64, the Atari ST, the Amiga, and the IBM PC understands how important portable code is, and how to design and write it. For developers who enjoy a challenge, writing truly portable code is an attractive one.

GUIDO HENKEL | *Hailing from Southern California, Guido is a veteran of the game industry. He has worked on dozens of game titles over the past 18 years, including the* REALMS OF ARKANIA *series and* PLANESCAPE: TORMENT. *Most recently he founded G3 Studios (www.g3studios.com), a developer and publisher of mobile games.*