gd

GAME DEVELOPER MAGAZINE

SEPTEMBER 2001

UBM Tech

GAME PLAN

LETTER FROM THE EDITOR

Input

I pulled out the box from my old Atari 2600 last weekend. It's the same box which filled me with a sense of wonder that afternoon late in September 1981. After spending an entire summer picking berries I had saved up enough money to purchase the latest in videogame technology. Examining the box last weekend I noticed the controllers. I'm sure you remember them — to describe them as elegant would be a stretch, but they were certainly simple, and rugged. One stick, one button. That's really all you needed.

And then I looked at my Playstation 2's controller. It really isn't much different. Twenty years later, I have more sticks and more buttons. O.K., the Playstation 2's controller has a pager-like vibrator in it to give some haptic feedback, and some analog controls, but really, that's about the sum total of 20 years of videogame controller innovation.

What happened to my speech recognition? Head-mounted displays? Motion-sensing trackers? Video-based interaction? Datagloves? Have you seen the movie *Existenz*? How about those hammocks from *Lawnmower Man*?

O.K., some of these aren't feasible for technological reasons. But truth be told, alternative input devices aren't available for two simple reasons.

Not Included

In general, people don't buy peripherals for their consoles. They stick with what came in the box, or perhaps they buy an extra controller that is the same type as the original. It doesn't matter what peripherals you ship after a console launch; if it doesn't come in the box, it is difficult to get serious market penetration.

Not Supported

Of course, if developers can't expect people to have a particular input device, why would they bother supporting it? As a developer, you have enough work to do. You're much more likely to spend time improving your game than ensuring that some alternative input device is supported.

The only special cases where new input devices have done passably on consoles are when new consoles begin shipping with the new devices so that developers can expect an installed base. A good example of this is the Playstation DualShock controller. Ever since Playstations began shipping with DualShock controllers, every new console to appear has included two analog sticks.

But this creates evolutionary change, not revolutionary change. Thus what you see when you look at a modern-day console controller is an evolved version of the original Atari 2600 controller. How do we create revolutionary change?

Nvidia's treatment of DirectX 8 vertex and pixel shaders shows a good model. They bring in game developers to educate them on how to take advantage of these new features. They give away free hardware. They educate the press. And they make sure that there are "launch titles" — games which take advantage of the new features — as soon as the hardware hits the market. In short, Nvidia has done the exact same thing for GeForce 3 that console manufacturers do in the months and years prior to launching a new console.

Input device makers need to take a lesson from Nvidia and the console manufacturers. In order for a paradigm shift to occur in the way we interact with games, they need to educate developers and players to create buzz, and then make sure that there are new games available which show off the new features.

Until then, we're all stuck with our sticks and buttons. Which is fine, but in another 20 years I'm going to be running out of fingers to twiddle things with, so someone better get to work on a foot joystick.

And I better start doing some yoga.

Mark

GameDeveloper

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

Publisher
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL

Editor-In-Chief
Mark DeLoura *mdeloura@cmp.com*
Senior Editor
Jennifer Olsen *jolsen@cmp.com*
Managing Editor
Laura Huber *lhuber@cmp.com*
Production Editor
Olga Zundel *ozundel@cmp.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Daniel Huebner *dan@gamasutra.com*
Jeff Lander *jeffl@darwin3d.com*
Mark Peasley *mp@pixelman.com*
Advisory Board
Hal Barwood LucasArts
Ellen Guon Beeman Beemania
Andy Gavin Naughty Dog
Joby Otero Luxoflux
Dave Pottinger Ensemble Studios
George Sanger Big Fat Inc.
Harvey Smith Ion Storm
Paul Steed WildTangent

ADVERTISING SALES

Director of Sales & Marketing
Greg Kerwin e: *gkerwin@cmp.com* t: 415.947.6218
National Sales Manager
Jennifer Orvik e: *jorvik@cmp.com* t: 415.947.6217
Senior Account Manager, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 415.947.6224
Account Manager, Northern California
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, Silicon Valley & Asia
Craig Perreault e: *cperreault@cmp.com* t: 415.947.6223
Sales Associate
Aaron Murawski e: *amurawski@cmp.com* t: 415.947.6227

ADVERTISING PRODUCTION

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

GAMA NETWORK MARKETING

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

Game Developer magazine is BPA approved

CIRCULATION

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

SUBSCRIPTION SERVICES

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

INTERNATIONAL LICENSING INFORMATION

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

CMP MEDIA MANAGEMENT

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

GamaNetwork
A DIVISION OF CMP MEDIA LLC

INDUSTRY WATCH

daniel huebner and jennifer olsen | THE BUZZ ABOUT THE GAME BIZ

Sega in transition. Sega is continuing its transformation from hardware company to software provider. The company reinforced its decision to focus on its core businesses by signing a number of European publishing and distribution deals that could signal Sega's departure from the European market. Sony Computer Entertainment Europe has announced a deal to publish seven Sega Playstation 2 titles in SCEE's European PAL territories. The deal also includes localization into five languages for titles such as ECCO THE DOLPHIN, FERRARI F-355, VIRTUA FIGHTER 4, two SPACE CHANNEL 5 games, and an as-yet-unannounced game code-named "K-Project." Infogrames is set to publish several Sega titles there for other consoles, including PHANTASY STAR ONLINE VERSION 2 and VIRTUA STRIKER 3 for Gamecube, CHU CHU ROCKET and SONIC ADVANCE for Game Boy Advance, and JET SET RADIO FUTURE and HOUSE OF THE DEAD 3 for Xbox. Sega handed off the remaining Dreamcast distribution for much of Europe to Bigben Interactive last April.

Games are not the only thing Sega has on its mind. The company is hoping to leverage its entertainment expertise to build revenue outside of the game business. While not yet ready to announce any concrete plans, the company believes that its computer graphics, content development, and arcade management expertise should present opportunities. "We have been told that Sega has a lot of hidden treasures," said Munehiro Umemura, general manager of Sega's future entertainment division, "We will dig them up one by one, and will offer them to our customers." New businesses might include educational and training software.

Two of Sega's SPACE CHANNEL 5 games for Playstation 2 are part of a European publishing and localization deal with former rival Sony.

Exits for Lucas Learning and Midway. Lucas Learning has cancelled its two remaining announced products in development and is pulling out of the consumer game market. The company has discontinued planned ports of STAR WARS SUPER BOMBAD RACING for Macintosh and PC, and will focus instead on creating and marketing a line of curriculum-based educational software directly to schools for use by students in grades K through 12.

Midway recently ended an arcade tradition dating back to 1975. The company officially announced its long-anticipated decision to halt all coin-op game development efforts. The company cited continuing decline in the arcade market as well as a desire to focus its efforts on home games for next-generation console platforms. Closing the coin-op business will result in layoffs of close to 60 employees and a pre-tax charge of $8 million.

Opening shots. With the biggest shots of the console wars just about to arrive, Sony made a preemptive strike by cutting the price of the Playstation 2 before the launch of rival systems. The price was lowered to ¥35,000 ($280) from the current ¥39,800 ($320). The lower price applied only to Japan, with Sony leaving it up to its subsidiaries to determine the retail prices for their local markets.

Nintendo of America announced that most of the 500,000 initial units shipped for the Game Boy Advance launch sold in a single week. Nintendo characterized it as the most successful game console launch in history. Sony disputed that claim, holding that the Playstation 2 launch was more successful.

Titus holds Interplay shares. French game publisher Titus saw its shares shoot up as much as 10 percent after the company announced that it would not pursue plans to sell its stake in Interplay Entertainment. Titus had earlier announced that it was in discussions that might result in the sale of its share of Interplay. Titus's holdings currently stand at 34 percent of Interplay's stock and 40 percent of the voting rights, making Interplay Titus's single largest asset. Titus also revealed that it might consider increasing its stake in Interplay.

Infogrames hands down Humongous layoffs. Eighty-two employees, more than 40 percent of the company's workforce, were laid off from Washington-based children's game developer Humongous Entertainment. A spokeswoman for Infogrames, which owns Humongous, said the company's remaining 117 employees will focus on the company's successful BACKYARD series of sports games for children. New titles based on popular Humongous properties like Pajama Sam and Freddi Fish will be produced only if the company sees increased market demand.

Head office changes. Former Xbox third-party relations director Kevin Bachus has joined web game technology maker WildTangent. While with Microsoft, Bachus also served as group product manager for DirectX and was an originator of the Xbox concept. At WildTangent, Bachus will take on the role of senior director of marketing.

Acclaim Entertainment has appointed David J. Sturman to the post of chief technology officer. Formerly vice president of software development for TheStreet.com, Sturman will oversee the development of Acclaim's proprietary engines and tools for next-generation titles, reporting to the executive vice president of product development.

Acclaim posts profitable Q3 results. Acclaim marked its third straight profitable quarter of its fiscal 2001 with a profit of $0.2 million on net revenues of $38.6 million in its third quarter, up from revenues of $33.8 million for the same period last year. Acclaim met much of its third-quarter profit through a combination of higher net revenues, reduced operating costs, and debt reduction.

UPCOMING EVENTS CALENDAR

IMX-THE INTERACTIVE MUSIC XPERIENCE
LOS ANGELES CONVENTION CENTER
Los Angeles, Calif.
October 9–10, 2001
cost variable
www.imxevent.com

PROJECT BAR-B-Q
GUADALUPE RIVER RANCH
Bourne, Tex.
October 18–21, 2001
cost: $2,150
www.projectbarbq.com

ALIAS|WAVEFRONT'S MAYA 4

by steve theodore

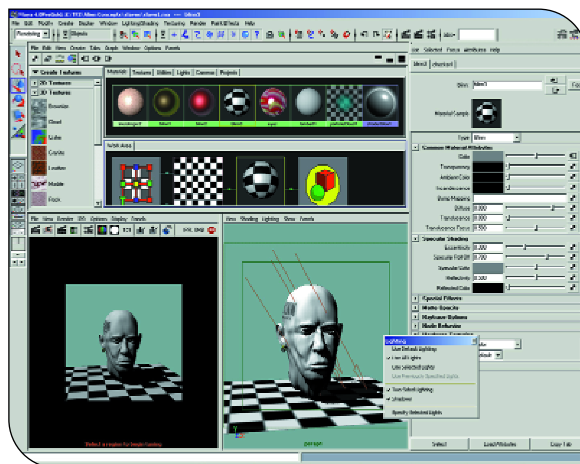
Alias|Wavefront bills Maya 4, the latest release of its advanced 3D modeling and animation package, as "the easiest ever." Although the upgrade offers relatively few major new features, Maya 4 sports dozens of workflow enhancements and refinements.

Workflow improvements. Maya veterans will notice immediately that the workspace has been rearranged. The basic tools have been moved to a vertical bar along the left side of the screen. This toolbar sports a new Lasso selection tool, similar to the 2D paint program standby. It's an example of what works best in the new Maya: a simple addition that really enhances the workflow. Other helpful additions include an alignment tool, similar to those in 3DS Max, and a new menu option for inverting a scene selection.

The viewports themselves have received some notable enhancements. The API offers a new class that lets users plug their own game renderer right into Maya's editor windows. Unfortunately, Maya supports only OpenGL-native rendering, so if your game engine uses DirectX, it may not be possible to get a perfectly faithful representation of your renderer within the viewports. Nevertheless, the new class is a big step toward eliminating unpleasant surprises from the production pipeline. Even without additional code, however, the editor windows can display GL distance fog. Spots and directional lights can even cast depth-mapped shadows in real time. Hardware texture support has also been enhanced with more control over filtering and both shaded and flat-lit display modes.

Modeling tools. Maya's highly regarded modeling tools received only tweaks in this version. The ability to texture patches within a NURBS surface is a useful feature, although the shader assignment, puzzlingly, is lost when the NURBS surface is converted to polygons. The polygon tools are basically unchanged; a new command to select a line of contiguous edges is handy for dealing with seams in symmetrical models. The addition of the ability to animate vertex colors rounds out Maya's excellent set of vertex lighting and coloring tools. While the performance costs may make it slow to animate vertex colors on dense meshes, it will be a welcome trick for teams dependent on vertex lighting.

Artisan tools. Artisan is Maya's powerful tool for using a pressure-sensitive brush to manipulate 3D objects. Artisan is known mostly as an elegant method of sculpting geometry, but it is also used for setting skin weights, applying vertex colors, and selecting components. Maya 4 makes Artisan tools fully compatible, at last, with lower-end GL hardware (particularly GeForce cards). The new architecture allows painting and mirroring of Artisan strokes in world space, where previous versions were limited to operations in UV space and were vulnerable to paramerization problems. Oddly, only about half of the tools use the new architecture in this release, and the sculpt tools are among those that do not. This seems like a case of deadline pressure rather than a design decision.



Maya's user interface has been extensively redesigned for version 4. Shown here are the viewport shadows and the improved filtering renders.

Texturing and UV mapping. Maya 3 introduced Paint Effects, a sophisticated 3D painting plug-in which enables users to paint either simple colors or complex procedural geometry such as hair or foliage directly into three-dimensional space. In Maya 4, Paint Effects brushes have been folded into the Texture Paint tool, allowing users to paint effects ranging from natural media to procedural hair, grass, and lightning directly onto a model. Texture Paint is not ready to compete with dedicated 3D paint software, though — painting performance scales directly with texture sizes. Textures that are 128×128 or 256×256 can be painted with some finesse (performance on my 933MHz Pentium III machine with a GeForce 2 card ranged from 50 frames per second to around 25). At 512×512, however, the frame rate drops by half, and only simple tasks such as painting out seams or making registration marks are worthwhile; textures above 1024×1024 are too cumbersome to work with at all. Image-based functions like blur and smudge are also slow and muddy. For touch-ups, feature registration, and painting masks for procedural textures, however, the tool is quite capable.

Maya's well-regarded UV editor, the Texture view, has been revamped for easier navigation. You can now merge individual texture vertices, where previous versions would only merge or sew UV edges. A handy new tool smoothes out tangled UV borders, so that folded and mangled UVs can be rationalized. Particularly for game applications where texture space is such a valuable commodity, Maya's excellent UV-editing tools are a strong attraction. Finally, an excellent little utility allows the export of UV map templates for texture painting with pixel-registered accuracy and no recourse to the Print Screen button.

Rendering. Maya's renderer has historical-

ly been plagued by odd texture filtering, particularly on nonsquare textures. The new renderer does a vastly better job, with cleaner bump mapping and much better handling of high-contrast borders. Another old glitch, whereby NURBS surfaces tended to "leak" in raytraced images, has been fixed. The raytracer now handles translucent surfaces much more realistically. Rendering is also somewhat faster. Alias|Wavefront claims a 5 to 10 percent speed boost for most machines, with more for Pentium 4s.

The rendering interface has gotten some needed attention as well. A welcome touch is the addition, at last, of automatic default lighting to scenes being rendered. Maya veterans will rejoice to see that the Render Globals option menu is accessible through a toolbar button. The Render Globals window, though still somewhat overloaded with options, now offers the ability to store preset combinations of rendering variables. Finally, batch rendering no longer interjects a file-save dialog before each render. The cumulative effect of these small changes really softens one of the only rough spots in Maya's interface.

Animation. This edition of Maya offers animators some long-desired options as well as powerful new tools. Maya can now display "ghosting" in the viewports, images of an animated object illustrating past and future positions. Maya 4 can also generate motion trails representing the path of objects through space and time. While neither of these display options lets you edit keys directly, they are helpful aids to visualization.

Animators will rejoice over the ability to animate skeletons from inverse to forward kinematics and back. While it's always been possible to combine FK and IK in Maya (something many packages do not support), it formerly required tiresome hand-alignment of the keys at each transition. Maya 4 automates the process with a single command that synchronizes the transition cleanly. Although it is sometimes necessary to recalculate the transitions after an edit, the process is painless and offers animators enormous flexibility.

Trax, Maya's nonlinear animation system, allows users to edit animated clips much as a video-editing program edits video footage. Maya 4 makes Trax more approachable by simplifying the creation of Character Sets — the nodes that collect animatable attributes into "actors" for Trax to choreograph. Character sets can now be created through drag-and-drop editing in the Relationship Editor or the Outliner. Trax can even auto-create characters on the fly. The rest of the Trax editor remains basically unchanged, although it's now possible to apply clips from the editor without needing to bring up a Visor window. Clips themselves now feature completely remappable timing, or "time warps": you can slow, speed up, or even reverse the timing of a clip without changing the underlying keys. Many users will be disappointed to find that time warps work only on clips, however, and cannot be applied to individual animation curves outside of Trax.

One much-touted animation feature turns out to be rather an anticlimax. Until now, Maya has only supported Euler ("XYZ") representation of angles. While Euler angles facilitate fine control of timing and help economize on the number of keyframes required to describe a motion, they are also vulnerable to gimbal locking and wobbly interpolations. Maya 4 now offers quaternion interpolation, which prevents gimbal locks and rotates smoothly between any orientation. This is particularly useful in Trax, where it discourages various kinds of Euler-induced mischief when blending between clips. Disappointingly, however, the new quaternions don't support the f-curve key tangents that control interpolation in Euler keys. This means the only way to control the speed of a quaternion rotation is by setting more keys, an inelegant solution unworthy of Alias|Wavefront's usual standards.

Character setup. Maya 4 greatly eases the burdensome task of binding characters to their skeletons. The elegant Paint Weights tool, which uses the Artisan interface to paint deformation weights directly onto a skin, can now interactively select which influence to paint without a clunky dialog box. It also offers the ability to export weight maps as editable bitmaps, allowing skeletal assignments to be copied, archived, and even pasted between similar characters. A new Weight Pruning procedure lets you eliminate vertex weights that are too small to be useful, thus improving performance and simplifying further editing. Buoyed by improved Artisan performance, the setup of complex skinnings is now almost pleasant.

Roundup. Alias|Wavefront made a commendable decision in the planning of Maya 4. Resisting the lure of "featuritis," the company has refined Maya's tools and added a welcome level of polish to an already powerful product. A few features do seem rushed — the barebones implementation of quaternions, for example, and the low performance ceiling on the Texture Paint tool. On the whole, however, Maya 4 reflects Alias|Wavefront's traditional high standards. Maya 3 users should be pleased with the upgrade, particularly the enhanced animation and texturing features.

Those who have found Maya too intimidating in the past, however, may not find enough in this upgrade to convert them. Maya offers sophisticated, highly flexible tools instead of one-button, single-purpose functions. While the interface is responsive for knowledgeable users, those unwilling to learn the nuances of the package may find it difficult to approach. For the time being, Maya may not be the ideal tool for an inexperienced team. Nevertheless, Maya's latest incarnation remains one of the defining landmarks for all 3D animation and modeling and continues to set high standards for the industry.

MAYA 4 ★★★★½

STATS

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

Price: Maya Builder: $2,995; Maya Complete: $7,500; Maya Unlimited: $16,000; upgrade pricing available on request.

System Requirements: Windows NT 4 (SP 4 or greater)/2000: Intel Pentium processor with 128MB RAM and a three-button mouse; also available for IRIX, Linux support forthcoming.

PROS

1. New 3D texture painting tool.
2. Can keyframe IK or FK on the same joint chain.
3. Simplified character setup.

CONS

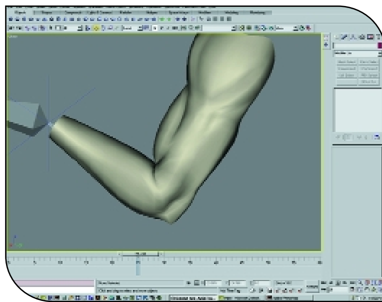
1. Inconsistent upgrades to Artisan tools.
2. Limited options on quaternion rotations.
3. Low performance ceiling for Texture Paint tool.

DIGIMATION'S BONES PRO 3

by stefan henry-biskup

Bones Pro 3 is the newest version of the long-popular skin deformation plug-in for 3DS Max. Appreciated for its ease of use and simple interface, Bones Pro 3 continues to fill a needed price niche in the Max tools world.

With skinning programs it is important to remember that the software's features fall into two categories if you are developing for games: those that can be exported to your real-time 3D engine and those that are only going to be usable for prerendered cinematics or sprites. Bones Pro 3 has some excellent features in both areas.



Meta Bone in action.

The adjustment of the bone influences over the model involves only two parameters, Falloff and Weight. They can be adjusted in real time with color-coding to show how much the bone will affect the mesh. In practice, the adjustment is smooth and performs well on a mid-level machine (600-MHz Pentium III with 128MB RAM and a 32MB TNT2 graphics card). Because the influence values are so simple, hand-editing of vertices is necessary to deal with unwanted influence in tight spaces. Luckily, Bones Pro 3 is equipped with a fine set of selection tools for getting at vertices and dealing with this. You can also more finely hand-weight the vertices to given bones. It is easy to add or remove bones from the skeleton and retain the settings for the bones you keep. File formats are provided to export the mesh and skeleton data in text format. The creation of a mesh weighted to a skeleton is fast and easy, just as the program intends.

In the higher-level skinning tools, things that are going to be used only for work rendered in Max, there are some very welcome features. New with this version of Bones Pro is the Meta Bone, a spherical primitive that can be linked to the skeleton and displace the mesh that lies around it. Examples that come with the program demonstrate how this can be used to maintain volume and bone protrusions beneath the skin easily on joints. The truly unique part of this functionality is that the mesh is allowed to slide over the volume of the Meta Bone. This is a very powerful feature that will allow effects no other program offers. There is also a plug-in that lets you turn any geometry into a Meta Bone, with my tests showing only slightly slower performance. The other major high-level feature is the Bone Jiggler, a secondary motion modifier that lets you add drag and bounce to the mesh to simulate weight and flexibility. This is assigned on a per-bone basis, so you can add bones strategically to get the effect only in desired places.

Bones Pro works best with meshes. It can be assigned to patches or NURBS but will turn both into meshes. It must be said that in running Bones Pro 3 through its paces, the software did lock up an otherwise stable machine a noticeable amount of times. Only the online manual was available, and it was barely sufficient. If you are familiar with these kinds of programs it's easy to pick up, but for a less experienced user it could be a little confusing. It would also be nice if the Meta Bone effect were stronger.

Bones Pro is old school, and for longtime fans of the program that has always been its appeal. It's all about mesh and bone. Simple and straightforward, without a lot of messy envelopes and such to clutter and confuse, getting a good setup is meant to be a breeze and it is. At $495, Bones Pro offers a more advanced skinning option than Max's built-in Skin modifier at less than half the price of Character Studio. If you plan on using the great new IK to do your skeletons, then it is a good option.

★★★★ | Bones Pro 3
Digimation | www.digimation.com

PROKSIM'S NET-Z 2.0

by crosbie fitch

Net-Z 2.0 is the latest version of Proksim Software's networking middleware designed specifically for games. It takes up where networking layer APIs such as WinSock, DirectPlay, and RTime leave off, focusing squarely upon synchronizing duplicated objects for games operating on Windows 95/98/NT/2000/ME, Windows CE/PocketPC, Linux, and Playstation 2.

Proksim has noted like few others in the games industry that there is a higher-level objective to address in enabling multiplayer games, and this distributed systems approach is designed to obtain a consistent or synchronized game state. Naturally, there are many aspects of the synchronization process that require knowledge specific to each application, and sensibly Net-Z has many features to address this.

So with Net-Z you're straight in at the level of thinking about how you're going to architect your distributed system, whether it's simple client/server, totally egalitarian peer-to-peer, or somewhere within the spectrum of those bounds. Thus, Net-Z allows you to control how object state is arbitrated, whether tied to a specific machine or spread across all or a subset of machines. Given that there is some burden to hosting an arbitrated object, or "duplication master" in Net-Z terminology, the policy by which this burden is shared is where Net-Z's load-balancing feature comes into play. You can either use a built-in policy ("equal object numbers" or "equal weights of weighted objects") or you can hook in your own if, say, some objects need a dynamic weighting.

Where Net-Z really demonstrates its next-generation credentials is in scalability. It can cope with ever greater numbers of players and an ever-increasing size of game world, without necessitating the use of expensive game servers.

In its earlier 1.5 release, Net-Z duplicated the game state across all participating computers. This meant that although Net-Z-based games were able to break through the bandwidth barrier of the client/server architecture, they were still significantly bound by bandwidth limitations. Although spreading arbitration across multiple computers greatly reduces the need for a dedicated high-bandwidth game server, the game state still needs to be synchronized (and 56Kbps isn't much). However, now Net-Z supports a feature called duplication spaces. This is an extremely useful and flexible concept that can be used to decompose a game world into relatively independent elements, called cells. This is better than imposing an ad hoc requirement as to how you partition your game world, such as into a cubic grid

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

(regular or octree), portal-connected rooms, or server-tied territories. While these ad hoc zoning mechanisms might still allow location transparency (zones could be mobile between computers for load balancing), they'd still exhibit a hard-wired nature, meaning a system oriented for a specific class of application.

A duplication space allows you (rather than the middleware vendor) to decide your own zoning policy, because it is an associatively computed mechanism. It could be based on spatial coordinates, it could be based on which room an object is associated with, and it could even be based on a dynamic spatial-partitioning scheme. You can have different schemes for different purposes; for example, duplication spaces for high scores or other sensitive player stats, duplication spaces for static scenery, and duplication spaces for dynamic objects. Perhaps visibility-based cells might be useful. Cells don't have to be exclusive, either; you can even have a multi-resolution duplication space with small cells intended for distributing across small computers, and medium and large super-cells for more capable computers.

Net-Z represents a C++-programmer-friendly integration of a distributed objects system into a game. There's still a big benefit if you understand what's going on down below, but it's nice to leave the mucky plumbing to someone else. However, if you do want to get your hands dirty, there's plenty of opportunity.

If I had to convert a game, or was familiar with a single-player game engine and had to use it to produce a multiplayer game before Christmas, then I'd be happy to use Net-Z. If I were writing a game from scratch, then it's probably a matter of economics and how scalable I'd want to go as to whether I'd opt for a full engine with networking built in, or whether to hook up an existing engine to Net-Z.

★★★½ | Net-Z 2.0
Proksim Software | www.proksim.com

PocketPC Game Programming
BY JONANTHAN S. HARBOUR
reviewed by curt tooley

The rising popularity of PocketPC (Windows CE 3.0 operating system) handheld devices has led naturally to a huge demand for games for this platform. PrimaTech's *PocketPC Game Programming: Using the Windows CE Game API* aptly fills the void of information about writing game applications specifically for PocketPC. The book is nominally targeted at a beginner/advanced level, which is a bit confusing. A better choice might be advanced beginner/advanced. The first third of the book contains information that is old hat to the more experienced coder. Still, newcomers to the PocketPC need to have some experience in coding and, most importantly, a familiarity with C++.

Author Jonathan S. Harbour and his editor André LaMothe have compiled more than 700 pages of clear, concise information on game programming for the PocketPC, offering readers a useful working knowledge of the PocketPC platform and its limitations. Harbour discusses programming requirements and offers incrementally developed sample code to help readers maximize the PocketPC's capabilities.

Following the introductory chapters introducing the PocketPC platform and Windows CE 3.0, the next chapters cover the use of Embedded Visual Basic and Embedded Visual C++, as well as the Windows CE Game API library, in clear detail. While it's not DirectX for the PocketPC, the Game API, or GAPI, enhances the PocketPC's ability to blit graphics (2D) to the screen much faster than using the built-in Windows CE GDI.

With clear and well-commented code samples that actually work the very first time, readers are able to see the fruits of their labor as they write and compile them. The samples help readers to build graphics and sound routines that, while rudimentary, illustrate the fundamentals of getting an idea from concept to LCD.

Readers also get a thorough treatment of the Embedded Visual Basic development system for game programming. Although Visual Basic is somewhat limited for game applications, readers can and do create a full game using the book's examples and the Embedded Visual Basic system. Experienced readers may already understand that, as a fully compiled system, Embedded Visual C++ is much more powerful than Visual Basic for creating game applications. Still, even those with no previous experience using Embedded Visual C++ soon feel right at home. Harbour knows his material and explains it well to those who don't. Note that a working knowledge of C++ is necessary to fully grasp all the information presented. Code samples in this section guide readers in the creation of functional games, as well as core game graphics and audio function libraries for use with the game code in the book and afterwards.

The final chapters of the book outline a few important advanced development topics such as war-gaming theory, computer controlled players, game physics, infrared and socket communications, and TCP/IP protocol for multiplayer gaming. As with earlier chapters, Harbour illustrates his points with working code samples.

The companion CD contains the Embedded Toolset (Microsoft's programming suite that includes both Embedded Visual Basic and Embedded Visual C++), the latest version (1.2 as of this writing) of the Windows CE Game API, along with all of the source code described in the book.

Good book? Yes. Perfect book? No. The selected commercial tools and their included demos on the CD are overly hyped in the book's text. The book includes helpful information on other software tools, but it seems that those available for purchase are given more space — it's as though the reader must suffer through a commercial in the middle of a lesson. Better to give readers a URL if it's available, or some other basic information, and be done with it. Furthermore, Harbour might have offered more details on using the Microsoft Embedded Toolset, and the lessons on networking multiplayer games could have been expanded upon.

Despite these minor shortcomings, at $49.99, the book is worth picking up for those who need a solid overview of PocketPC game development. No matter your level of experience you will find this a welcome addition to your reference library.

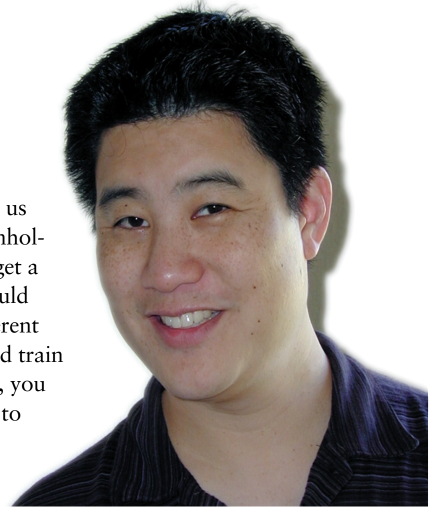
★★★★ | *PocketPC Game Programming*
Prima Publishing | www.primapublishing.com

PROFILES

jennifer olsen | TALKING TO PEOPLE WHO MAKE A DIFFRERENCE

Cyrus Lum

Drawing on the Past



ABOVE. Inevitable Entertainment's Cyrus Lum

Cyrus Lum is an elder statesman in the world of game art. He began his 12-year career at Strategic Simulations, and later established and ran art departments at Crystal Dynamics and Iguana Entertainment (later Acclaim), where he worked on such hits as TUROK and the QUARTERBACK CLUB series. In 1999, he and two fellow VPs from Iguana, Russell Byrd and Craig Galley, founded Inevitable Entertainment, whose first game, a version of TRIBES 2 for Playstation 2, debuted earlier this year at E3.

Game Developer. What was it like back in the olden days at SSI?

Cyrus Lum. Back then, everything was 2D, and then this small program came out called 3D Studio. So I started playing with it. 3D Studio was this simple CAD program that you could actually start to do some animation with. You could set some keyframes and have some renderings come out that actually looked pretty nice. I was trying to convince SSI, "Hey, you know it would be great to get some long-format animation, it could be like a little movie!" Nobody had heard of FMV or cinematics or anything like that, so they were sitting there like, "Yeah, that'd be cool, but we don't have that much room on these 5-1/4-inch floppy disks."

GD. You were offered a job at ILM at one point. Tell us about that.

CL. When I was ready to leave Crystal Dynamics, my biggest struggle was trying to decide between Iguana and ILM. A lot of my friends were saying it was an easy decision — ILM, of course! But I sat down and I thought about what it is that I really enjoy about doing computer graphics. And to me it's having that creative input. At ILM, even though you'd be able to get your name in the lights and it's exciting, you're more like a contractor; it's like a print shop. If I want creative control, I want to make the decision as part of the production company, the people who actually contract ILM. I was thinking I would really miss that a lot. With games, if I need to change something, I can. That was the thing that told me right there, "O.K., Cy, I think you want to stay in games." You've got to follow your heart and not the hype.

GD. You set up at Iguana a unique environment for game artists to learn and work in. Explain a bit about that.

CL. What I was offering was kind of like the Army of computer graphics: "Hey, you wanna drive a tank? We'll let you drive a tank!" (Laughs.) We'll train you! When I came up with that idea, a lot of my other friends in the industry thought that when you train people, they're just going to leave for better opportunities. But I thought if I can do the training and also create an environment that allows the artist to feel like they're growing, maybe that would allow me to keep people. So I developed this system of career development. In order to grow in the department, education was the single most important thing. When you first come in, we will educate you. As you start to learn and understand, in order for you to grow to the next level, you need to educate someone under you to get them up to your level. It allowed us to get away from pigeonholing artists. I wanted to get a system where people could actually migrate to different positions, so if you could train someone to replace you, you could actually move on to something else. Knowledge didn't become a job security thing. If you can help to get other people to understand, that tells me that you understand what you're doing, and you have the right attitude for being open to helping other people grow.

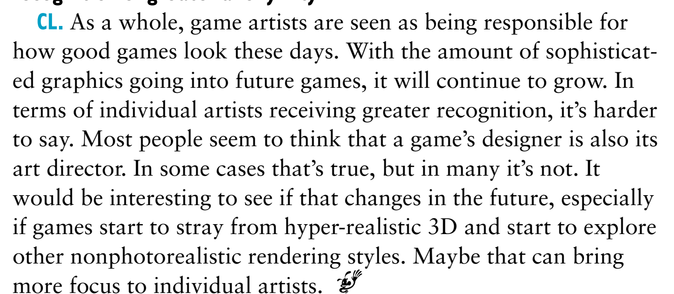
GD. So you were successful in retaining your art staff that way?

CL. A lot of friends that I know, I talk to them every year and they're at a different place. And it always happens right after Siggraph. The first year we went, I took everybody and I came back with the same number of heads. And I thought: this is a good sign. Five years later, only two people had left. A lot of those people came over with us to Inevitable, because of the great consolation prize of providing an environment for people where it's not like, "Oh, I'm just giving you a job and that's it." I'm giving you a job, I'm growing you, I'm investing in you. People understand that, and the loyalty that gets generated is very important.

GD. You've seen all the 3D packages evolve over the years. What are they still lacking that you wish they offered to game artists?

CL. Per-pixel shading is the big thing that everybody's interested in, but there's not really an interface that allows an artist to get in there and try to play with it. A per-pixel shader really needs a programmer sitting there and the artist talking to the programmer. But we need some kind of interface that has some generic per-pixel code that an artist can try out through sliders or something. There are a lot of cool things you could do with it, but there currently aren't the tools available for artists to take advantage of that.

GD. Do you think game artists are on their way toward greater recognition or greater anonymity?



CL. As a whole, game artists are seen as being responsible for how good games look these days. With the amount of sophisticated graphics going into future games, it will continue to grow. In terms of individual artists receiving greater recognition, it's harder to say. Most people seem to think that a game's designer is also its art director. In some cases that's true, but in many it's not. It would be interesting to see if that changes in the future, especially if games start to stray from hyper-realistic 3D and start to explore other nonphotorealistic rendering styles. Maybe that can bring more focus to individual artists.

The Life of a SILICON Stuntman

FIGURE 1. A character model demonstrating different action poses.

Growing up, my friends and I were hooked on action shows. I used to powerslide around in my parents' station wagon like it was some souped-up car from *The A-Team*. We would all practice sliding over the hood for a quick getaway like Starsky and Hutch did on their Gran Torino. I am certain that if we could have figured out how to build one of those jump ramps from *The Dukes of Hazzard*, we would have done it and probably killed ourselves.

Back then, we all wanted to be stuntmen. These artists get all the excitement and action without having to actually act or learn boring dialogue. So we practiced. We bumped, jumped, fought, and fell. Bloody broken noses, sprains, and scrapes were worn like a badge of honor. One time my friend, after attempting a particularly challenging (some would say stupid) trick on his BMX bike, flew over the handlebars and busted his chin on the curb. He ran home to have his mom take him in for stitches. We could follow his blood trail home for the next six months, and as the months passed, the story just got better and better.

These days, I am content to sit back and watch Jackie Chan break bones and Steve Irwin bleed profusely from various animal bites. I still get the same thrill without the risk of breaking my nose again.

What I Really Want to Do Is Direct

As a game maker, I would love to be able to direct my actors to perform the kinds of stunts that any good action sequence requires. I would like to take some of my virtual actors and put them in a good location such as a barroom, and then direct one to start a knock-down-drag-out bar brawl. You know the kind. A room filled with chairs breaking on backs, bodies flying across the room, and the barman wisely ducking behind the oak bar.

To create this scene, I could stick some animators on the job and animate the entire sequence just as I envision it. However, I want an interactive game, not a movie. I want a character to be able to enter the situation and change the outcome. Characters need to react to the situation, both dramatically as well as in a physically plausible manner.

Last month, I began to apply some technology to deal with the physically plausible part. I created a physical representation of an actor that could be used for some limited dynamic simulation. However, I would like to take a minute to look at the animation side of the problem by describing my character animation system.

With hierarchical character animation, the animator creates a target position and rotation for each bone in the actor's body. The actions can be either animated or static. In general, each action would be a single move such as walking or sitting, as you can see in Figure 1 from Curious Labs' Poser program. This program is a lot of fun for trying out a variety of different poses and actions on different characters. As a bonus, you can save out your action as a .BVH motion capture file that you can use for your real-time characters.

I could create actions for every task that I want my actors to perform and then just play them back when needed. However, if my character is going to be very interesting, that could wind up being quite a few actions. The actor can have a much larger range of actions if I combine the basic actions in order to create new actions. This technique has been used very successfully in games such as HALF-LIFE and SOLDIER OF FORTUNE.

Character actions can be blended in a similar way to keyframe interpolation. When an animation is created at a fixed rate, say 15 frames per second, and the playback of the game is much faster, say 60 frames per second, the animation can look slow and choppy. That is because there are not enough frames to make the animation look smooth. However, it is possible to calculate in-between frames that will smooth out the animation. I, as well as many others, have written about the use of quaternions to calculate these in-between frames of animation (see For More Information). You can accomplish this by using spherical linear interpolation (SLERP) to find a quaternion that is a specified distance between two existing quaternions. That is:

```
result = slerp(a,b,u);
```

where `a`, `b`, and `result` are quaternions and `u` is a value from 0 to 1.

This technique works very well for blending between two frames of animation. You can also use the very same blending technique to

JEFF LANDER | *When not sitting on the couch watching extreme sports or old seventies cop shows, Jeff directs the action at Darwin 3D. Send your favorite stunt story to him at jeffl@darwin3d.com.*

generate new animations. Consider the two character poses in Figure 1. If the orientation of each bone in each pose was represented by a quaternion, I could use the SLERP function to blend between the two poses. This would be very similar to morphing between two vertex meshes, except in this case, the only thing changing would be the orientation of the bones in the skeleton. This skeleton could then be used to deform the vertices of a skin mesh.

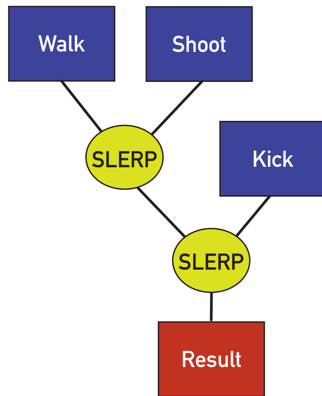


FIGURE 2. The motion mixer.

I like to think of this mathematical mixing of motion data as a fader that allows you to gradually fade between two actions dynamically. Of course, this technique can be expanded almost limitlessly. You can feed the results from one SLERP operation into the arguments of another blend. In much the same way that a recording engineer can combine a variety of inputs and mix them together to make something unique, you can now combine animation data of all types to make unique actions. You can see a flowchart of this mixing system in Figure 2.

One interesting application of this mixing motion idea is the creation of a pseudo inverse-kinematics system. Think of a character aiming a weapon. The artist could create animations for aiming up, down, left, and right to the maximum extent that the character can reach. By mixing the left, right, up, and down actions and combining the results, you can make the character aim at any position within reach of the various actions. Now, of course it may be much easier to use a simple iterative or analytic IK function to do this type of simple aiming with look-at constraints. In fact, an algorithmic IK solution can be mixed in to one of the SLERP functions just as if the artist had created the action. In practice, I find this very useful for making an algorithmic motion look more natural.

The basic motion mixer that I have described so far is not adequate to create a real animation system. There is still a big problem. In the process of mixing the animations, every joint is currently mixed equally. This won't produce the desired effect. Imagine combining a "walk" and a "shoot" action as I have just described. The "walk" action would involve the legs moving back and forth and the arms swinging along the sides. For the "shoot" action, the character is probably standing still and raising his arms to aim.

As I have described the motion mixer so far, if you start with the "walk" animation and then blend in 50 percent of the "shoot" action, instead of a walking and shooting animation, I will get a really bad walk with a very shaky aim and shoot. That is because the motion mixer is doing what I said, not what I wanted. When I tried to combine the walk and shoot actions, what I wanted was a motion where the legs keep on walking and the arms begin to move to the shoot position.

There was some information implicit in the shoot action of which the system was unaware. The action of shooting in this example simply involves raising the arms to aim and shoot. This particular action really doesn't involve the lower body, though there may be a small amount of recoil to deal with if the weapon is very large (as it always seems to be in most action games). Likewise, an action such as the "kick" in Figure 2 would probably largely involve the legs only, so you may not want it messing up the arms from the "shoot" action.

The solution to deal with these issues is to introduce a weighting system into the motion mixer. For each bone in each action, there is a weight value that describes how important each particular bone is to the action. You could probably calculate this weight value automatically by looking at the amount of change in each bone orientation throughout the action. However, I find it works better to set these weight values manually in some manner. That way, you have complete control over what parts of each action you blend into the final motion.

This weighted motion-mixing system is adaptable enough to provide a great deal of flexibility while still maintaining control over the actual performance of all of the actions. However, even with all the motion mixing, these actors have a pretty limited repertoire. When it is time for a physically challenging action such as falling off a ledge or tripping over a table, these actors may not be up to the task. Here, the virtual stunt team can come in and earn their pay.

Quiet on the Set

Using the technique I described in last month's column, "Last Call at the House of Blues" (August 2001), I can take a skeleton from a kinematic character and build a physical representation. In creating this representation, I use some information about how I want the different joints in the object to move; for instance which joints are going to be 1D and which joints need a full 3D range of motion. You can see the physical representation of my current stuntman in Figure 3.

The stuntman is composed of a series of points connected by rigid links. The points are assigned masses that approximate the relative mass of each part of the body so it will behave realistically when being simulated. There are also soft constraints consisting of springs that try to keep the joints of the body within the motion limits of the character.

As far as stuntmen go, this guy is pretty bad. He has no sense of balance and can't even walk. He apparently feels no pain, either, since if he falls on his face, he will not put his hands out to break his fall. However, the falls are pretty dramatic and provide a great deal of variety.

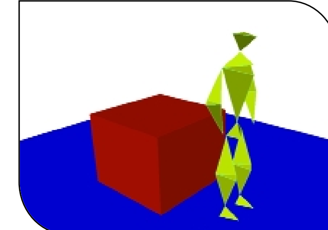


FIGURE 3. The stuntman arrives on the set.

Just like a real stuntman, this system steps in when the animation system is unable or unwilling to perform an action. For example, if I were to hit my character in Figure 3 very hard across the jaw, I would expect the character to fall back onto that box behind him. However, since the char-

acter might have been anywhere when he was hit, I probably don't have an animation ready where the character falls over and drapes over a box.

So the game detects when something interesting has happened, such as when a character has hit another character hard in the jaw. The motion mixer stops and the virtual stuntman assumes the exact pose where the actor was standing. Clearly, the stuntman needs to be the exact same size and have the same skeletal system as the actual game actor.

The game's dynamics system is then turned on and a set of forces is applied to the stuntman. These forces are a combination of external forces such as the punch and any internal velocities that have been generated by the animation system. This is accomplished by comparing the current bone positions with the last animation frame position.

Now the dynamics system takes over the animation of the character. Gravity pulls the character down and the force of the punch pushes him backwards. In order for the character to stop, I will need to do a bit of collision detection with the physical representation of the character. The floor is pretty easy, but creating a collision object for the table is going to take some more work. A table is a good candidate for a bounding box. I could make my life easier by requiring that all collision boxes be aligned with the world axis, but that would be a bit too restrictive.

So I will allow the character to collide with arbitrary bounding boxes. This is done by using the separation plane method that I discussed in "When Two Hearts Collide" (Graphic Content, February 1999). This method works on the premise that if I can find a plane that separates the stuntman from the box, the objects are not colliding.

For each face on the box, I take the normal to that face (stored or determined by the cross product of two edges) and one of the vertices of the face. This gives me the point-normal form of the plane of the face. Then, for each vertex in the stuntman's collision geometry, I create a vector by subtracting the point on the face. The dot product of this vector with the face normal will determine whether this vertex is outside the box. If it is not, the other five planes of the bounding box are checked. If a separation plane cannot be found, this vertex is colliding and I can resolve the collision by moving the vertex just outside the box on the nearest plane.

So algorithmically, the method for checking the collision with a box is:

```
for (each vertex, v, in collision geometry)
{
   colliding = true;
   for (each bounding box plane normal, n, and vertex, pv)
   {
         if (dotproduct(pv - v,n) < 0)
               colliding = false;
   }
   if (colliding)
         resolve collision for v;
}
```

Using this system, no vertex in the collision mesh will end up inside the box, and the link constraints will keep the body from falling apart. So what I get is the goal of a dynamic character that will fall down over a box. You can see this in Figure 4.

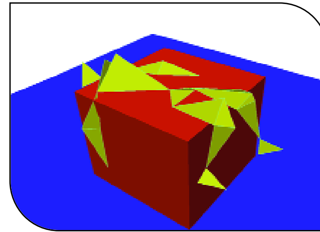


FIGURE 4. He fall down, go boom.

You can see the problem with this collision system. Even though the vertices of the stuntman are not penetrating the box, one of the edges of the stuntman's collision mesh is going right through it. It is obvious that the vertex test alone isn't enough to make sure there are no penetrations. I need to check the edges as well. I can go through each edge in the collision mesh of the stuntman, take the cross product with an edge on each face of the cube, and take the dot product of that vector with the cube face normal. If that test fails for all six faces of the bounding box, the edge is colliding and must be resolved.

Are We Ready to Wrap?

With the new collision detection in place, the stuntman falls in a physically plausible manner. It integrates well with the full motion-mixing system for kinematic animation. The dynamic animations do look more believable than generic falling animation. However, the stuntman still looks pretty dead. It's as if the instant the character is hit, he falls over either dead or completely passed out and nonresponsive.

In order for the character to look more like an actual stuntman, I need to make the character react to the fall. Blending in a little bit of generic "loss of balance" action may help quite a bit. I can also fire off a new action once the dynamics system has come to a rest, but we really need some intelligent dynamics in the stunt system. My stuntman should understand how to regain his balance and attempt to break his fall.

Unfortunately for me, that problem is a very difficult one that has stumped many of the best robotics researchers in the world. However, that doesn't mean we in the game development community shouldn't keep working on it. But not until next month.

FOR MORE INFORMATION

ARTICLES

Lander, Jeff. "When Two Hearts Collide" (Graphic Content, February 1999). www.gamasutra.com/features/20000203/lander_01.htm

Lander, Jeff. "Slashing Through Real-Time Character Animation" (Graphic Content, April 1998).

Shankel, Jason. "Interpolating Quaternions." In *Game Programming Gems*, edited by Mark DeLoura, 205–213. Rockland, Mass.: Charles River Media, 2000.

POSER 4

Curious Labs

www.curiouslabs.com

TILED TERRAIN

In this column, I'll examine some of the issues you may want to consider when starting to create a terrain tile set for a game. When done well, the terrain becomes a believable world in which your characters come to life. The user will simply accept your work as part of the natural backdrop. Often, the best compliment for a good environment is that it doesn't call attention to itself.

As the terrain artist, you may not have a ton of texture memory space to deal with, so it's always a good idea to figure out how to stretch your budget as far as possible. Obviously, the larger the number of base texture tiles, the more random (and hence natural) the terrain will look. When only a few terrain tiles are available, a repetitive pattern becomes very obvious. The trick is maximizing the usage of each texture while minimizing the disjoint that occurs when you see the same pattern over and over.

The Basics

Before you begin, you will need to evaluate the needs of your particular project. Let's assume that you are trying to make one large plain of cooled lava stone. In theory, you can accomplish this with just one texture map tile and one large polygon. You can tile the texture map multiple times on the single polygon, which gives the appearance of a higher resolution on the terrain. While this method works conceptually, it can quickly lead to some technical problems, such as the inability to add additional texture variety into the field, and shading issues over a large polygon.

Another method is establishing a grid of polygons similar to a chessboard mesh. You can map each square with a different texture map, and there is a lot more flexibility with what you can do on an individual square basis. This method addresses some of the variety and shading issues as well as making the process of editing and tweaking much easier.

Now, suppose you want to add a lake of molten magma to the middle of the landscape. With the single large polygon method, you'll quickly face some problems. Since you can't interrupt the tiled terrain and insert a different texture map in the middle of the sequence, you have to figure out an alternative method. One such alternative is to include the lake in one large texture map. While this would allow you to create a completely custom terrain, you will quickly encounter texture memory usage and texture resolution challenges.

The other alternative is to make a custom polygon for the magma only, and then perform a Boolean operation on that polygon, combining it with the other terrain polygon. This too presents some particular problems if you are using vertex shading to light your terrain. In addition, making the seam between the two texture types invisible will prove to be difficult, if not impossible.

For this tutorial, I'll focus on a more forgiving method of maximizing the effect of your tiled terrain by using a standard grid and texture-mapping each polygon quad or set of quads as a unique element. This method will allow you to control the layout of the textures much more accurately, and it also provides vertex-shading advantages. In addition, when you start to add vertical information to your mesh, you will find that the additional vertices give you more control of the 3D aspects.

Creating the Basic Flood Fills

When making a texture tile set, you'll need to create a minimum set of tiles. The complexity of the planned environment will determine the number of tiles, how well the tiles appear visually without showing repetitive patterns, and how many different types of random tiles will be required. The amount of available memory for the terrain will impact all these factors. As a good starting point, create the minimum base set, then add to it only when necessary. Reuse is king, so try to stretch your texture budget as far as possible.

The number of different terrain types you will have in your environment will be another big impact on your tile set. This impact can be compounded by which terrain types can touch or transition with one another. Take, for example, a terrain that consists of water, sand, and grass. If water can only touch sand and sand can only touch grass, then you only have two transition sets to make. However, if water touches grass as well, then an additional transition set is required. As you can tell, planning out the environmental requirements ahead of time makes a lot of sense. You may find areas where you need to limit the number of transitions in order to keep the texture budget in check.

For the sake of argument and to keep

MARK PEASLEY | *Mark has been in the gaming industry longer than he can remember. He's currently working on Xbox titles at Microsoft, and when not bleeding on the cutting edge, he's increasing his couch and mouse potato skills. His web site is www.pixelman.com, and he can be reached at mp@pixelman.com.*

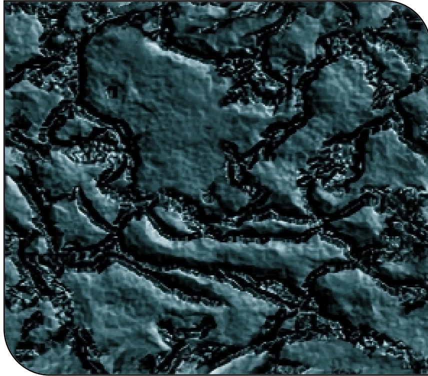


FIGURE 1. A basic stone texture that is seamless in only one orientation.

the number of variations low, I'll assume that the new terrain will consist of two basic ground types: stone (cooled lava) and molten lava. This will require you to make only one transition set. For now, don't concern yourself with the 3D aspects, and assume that the terrain is a flat plain.

First, create the flood-filled texture for each basic ground type. This is the basic tile that is used as the default flood fill of an area. If possible, since it stretches your texture budget, it's a good idea to make the texture able to tile in any direction, even if it's rotated 90 or 180 degrees. This will give you more visual mileage out of a single texture by breaking up the repetitive pattern of the texture when you lay it down next to itself.

In Figure 1, you can see the stone texture in its original orientation. It's easy enough to use the Offset filter and Rubber Stamp tool in Photoshop to make a texture seamless, but you're limited to only one orientation. If you rotate it and try to place it in a grid, the seam will be plainly visible. In order to make a tile seamless when it rotates, the texture needs to have identical edge pixels on each of the four sides. This requires a bit more work than just using the Offset filter, but it also extends the usability of the texture. With a bit of Photoshop magic, you can create the seamless effect pretty easily.

Making a Simple Tileable Texture

The first step in making a tileable texture is making your flood-filled texture tileable. There are quite a few methods and some programs devoted exclusively to this process. Photoshop is more than adequate for the job, so I'll cover a method that doesn't require any special plug-ins.

After obtaining the proper source material, choose a square section of the image. Avoid source materials that have a strong light direction embedded in the image. This is mainly because once you rotate and place the texture next to the original orientation, there is a visible anomaly in the lighting direction. If you take care, you can make strongly highlighted textures work, but that takes a bit more Photoshop work.

In this tutorial, I'll actually use a source that has a fairly strong highlight to make it easier to see the seams. Be aware of the image scale once it is mapped onto the terrain. It's very easy to find games out there where the texture maps aren't to the proper scale for the characters or the environment. Often, designers use the character texture maps to drive the texture resolution for the rest of the environment. Your main goal is uniformity in your texture resolution. The pixels per foot of your game should remain fairly consistent.

Once you have selected your perfect 256×256 texture, it's relatively easy to make it a seamless texture when no rotation is involved. To do this, load up the texture in Photoshop, then go to Filter>Other>Offset. Set the Horizontal and Vertical settings to 128 with the Undefined Area set to Wrap Around. The seam is visible, and you can then remove it using the Rubber Stamp tool. Try to keep from blurring the image too much, as it tends to make the end result look fuzzy. It's also a good idea to change the offsets to different settings such as 64 vertical, 64 horizontal and then check the seam again for visible anomalies.

Once you are done, check to see how it looks in a tiled environment. First, make a new Photoshop file, with the size set to 1024×768. Now open up the tiled texture you have just created and select all. Go to Edit>Define Pattern. This stores the texture as a repetitive pattern in the clipboard. Now switch to the 1024×768 image and click on the Paint Bucket tool. Go to the Paint Bucket Options panel and select Pattern from the Contents pull-down menu. Then, simply click anywhere in the image to fill the area with the pattern.

This process gives you a good idea of how the image will tile and whether there are any areas that tend to stand out in the repetitive pattern. Fix any obvious problems immediately rather than waiting until later. You'll have to redo most of the steps after this one if the edges don't quite work.

Additional Flood-Fill Patterns

After you create the first flood-fill tile, you can quickly make additional matching tiles. You may find that it's necessary to create three or four different flood-fill patterns to make the terrain random enough.

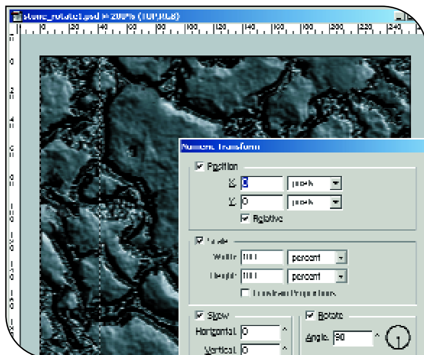
To create additional fills, you will need more source images. In this case, the original texture was a fractal pattern generated in Corel Draw's texture program, so by simply changing the seed number, you'll be able to make additional source material images all with a very similar look and scale.

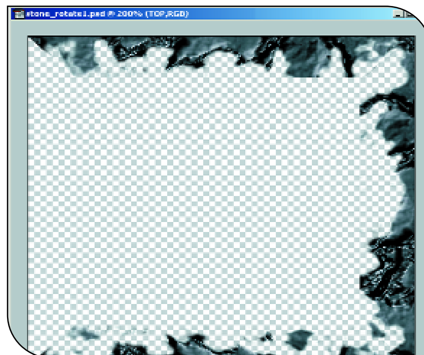
Once you obtain the new source image, create a new Photoshop file that has the new terrain as the first layer (I named mine New Stone) and the tiled texture you previously made as the top layer. Now, with the top Original Stone layer, make a selection box that is approximately 30 pixels from each edge. Accuracy isn't critical here, since you will be able to edit your work after the fact. With the selection still active, go to your Layers panel, and select the Add Layer Mask icon on the bottom left. This creates an active layer mask linked to your New Stone layer, with the edge border showing the Original Stone layer.

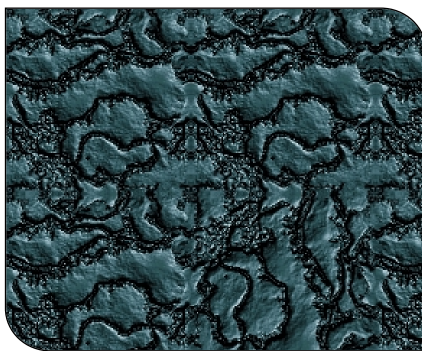
Now it's time to do some Photoshop work and edit the mask to blend the New Stone edge into the Original Stone edge. Don't edit all the way up to the edge. If you do, your new texture won't match the old texture at the seam. Once you complete this task to your satisfaction, save off a copy and collapse it down to create your new tile. By inserting it into your 1024×768 tile sample, you will be able to see how it looks and how effectively it breaks up your patterns.

Rotating the Edge

After examining the results of the two or three flood-fill textures, you'll







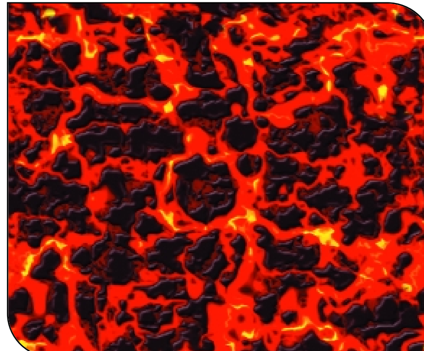


FIGURE 2 (above left). A section of the left border is selected and rotated 90 degrees. FIGURE 3 (above right). After the same section is placed on the three sides of the texture, its edges are erased to allow the texture below to show through. FIGURE 4 (below left). A test flood-fill pattern with the rotatable texture utilized in various orientations. FIGURE 5 (below right). A secondary texture is created that is both seamless and rotatable.

begin to recognize the pattern unless you've used several flood fills. Another way to extend your texture budget is to use a common edge on all sides. This method allows you to rotate the texture 90 degrees in any direction and it will still match. This technique also makes it much more difficult for the viewer to detect the pattern and requires a bit more Photoshop expertise to create.

First, pick a side of the texture map that will become your common edge. In the case of the stone texture example, I'll pick the left edge. Create a selection box that is flush with the top, bottom, and left side of the texture map, and about 30 or 40 pixels wide (see Figure 2). Now copy that piece of bitmap into the clipboard and paste it down again (Control-C then Control-V). This will place an exact duplicate of the texture section right over the original. Now make a selection set of the layer you just placed down by pressing Control and clicking on the layer in the Layer panel. Rotate the selection 90 degrees clockwise by right-clicking on the selection set, then choosing Numeric Transform and inputting 90 in the Angle section under Rotate. Now click and move the selection set so that it aligns with the top of the texture and name it Top. Add a layer mask, and edit the inside edge of the bitmap to blend it into the underlying texture map.

If you paste again (Control-V), you will get a new layer with the original left-hand side segment that was residing in the clipboard. Right-click on the segment, and select Free Transform. Right-click once again, and select Flip Horizontal. Now press Enter to lock the transform into the bitmap. Align with the right-hand side of the texture map, add a layer mask, and blend the inside edge to the rest of the underlying texture. Be careful not to remove the corner pixels. Rename the layer Right. Now select the Top layer and drag it into the Create New Layer icon on the Layers panel. This will create a duplicate of the Top layer. Rename it Bottom, then select and apply Free Transform. Choose Flip Horizontal, and you're done with the needed transforms. Now align it to the bottom edge and add or subtract from the layer mask layer to blend it into the underlying texture. Once you have done all three sides, you should have a rotatable texture with a seamless edge. Figure 3 shows the blended edges without the underlying texture for clarity.

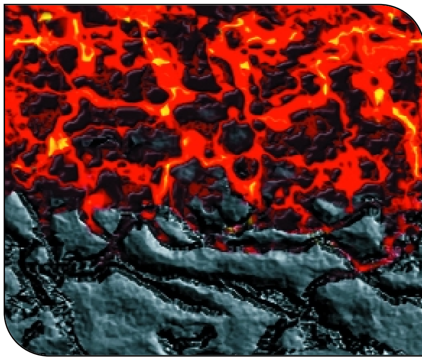
It is a good idea to test your new texture by creating a duplicate of the file and collapsing it down to one layer. Then run the Filter>Offset on it with 128 horizontal, 128 vertical and Wrap Around in the settings. This will make the nonrotated seams visible if there are any anomalies. Make sure either to undo or to run the Offset filter on it again to return the texture to the rotatable seam edge.

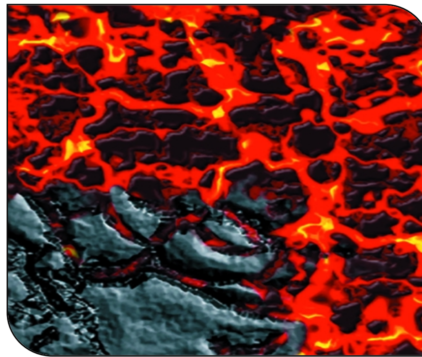
Now, select the texture and create it as a fill pattern. Create a new 1024×768 file and fill with the repeated pattern. Set the guidelines or grids on the 256-pixel boundaries and then paste a single texture into one of the grids as a new layer. You can now rotate the new layer 90 degrees and align it with the grids or guidelines. If you have done the steps correctly, it should blend in with the seamed edge without a visible line (see Figure 4).

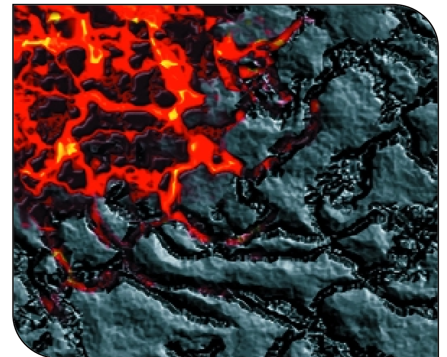
In order to create a blended set, we will also make a molten lava flood-fill texture, utilizing the same techniques I've just listed. The base lava texture was generated in Bryce 4, then run through Photoshop for a bit of image enhancement (see Figure 5). Once you have a tileable, rotatable lava tile, you are ready for the next step.

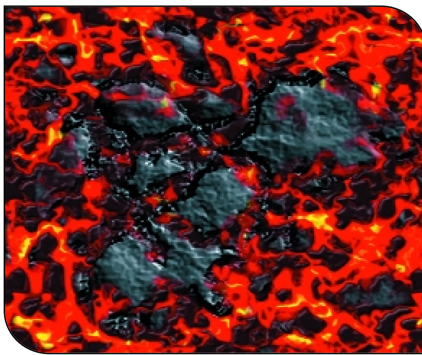
The Blend Set

At the most basic level, there are only three additional textures needed to create a blended set. The entire set contains five tiles: two flood fills and three transitions. For clarity's sake, I'll refer to them as the one-fourth blend, the one-half blend and the three-fourths blend. With these and a tiled environment, you can create any sort of varied coastline required. However, it's good to note that making three or four variants of each of these textures will make the transitions from one texture to another much less visible and more natural-looking.









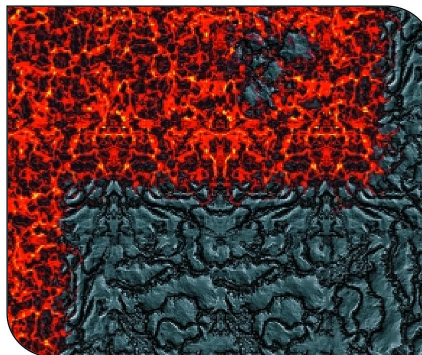


FIGURE 6 (upper left). The one-half blend texture. Note that the transition point on the edges is at exactly the 128-pixel point. FIGURE 7 (upper center). The one-fourth blend transition texture. FIGURE 8 (upper right). The three-fourths blend transition texture. FIGURE 9 (lower left). A unique texture that can be utilized (sparingly) to break up the patterns of the flood fill. FIGURE 10 (lower right). An example of the basic set of texture types and transitions used to blend between two terrains.

First, make the one-half blend tile. To create this tile, place the stone texture on Layer 1 or the Background layer in a new file you've created in Photoshop. Next, open and copy the lava texture into the clipboard. Paste the lava into the new file as a second layer over the stone.

The next step is either to add guidelines or to set your grid to show you where the 128-pixel point is on the side of the texture. Once you do this, select the top half of the texture, which should be a 256×128 selection box. On the layer mask, fill the selection with black to make it transparent and hide the texture under the selection. Now it's time to go into the texture and add some randomness. In the case of the stone texture, I tried to follow some of the natural contours of the rock. By alternating between black and white on the layer mask, I was able to add or subtract stone to the composite image as needed (see Figure 6).

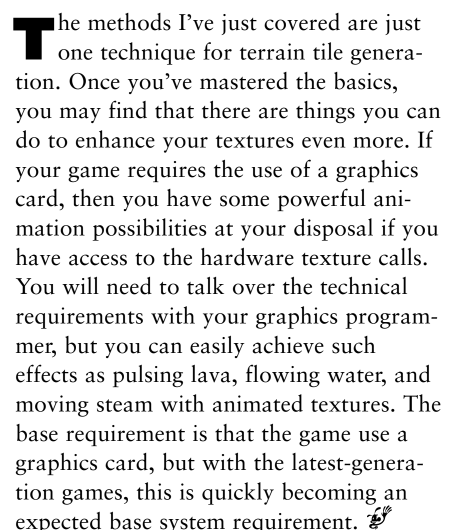
Avoid eliminating or altering the pixels at the very edge of the texture on either side. If you do, you won't have a tiled texture anymore. Also, try not to get too dramatic in the uniqueness of the transition. If, for instance, you decide that a big pool of lava would look good right in the middle of the transition, you will find that the distinctness of this element becomes apparent when the tile is repeated. If you are using multiple variations, then a unique tile every so often works quite well. The base repeating tile should be somewhat generic so that it doesn't bring attention to itself.

For the one-fourth blend and the three-fourths blend, take the exact same steps using the layer mask, editing only one quarter of the texture. The quadrant you choose is arbitrary, since the texture is rotated to allow for all four directions. Figures 7 and 8 show the final edit on the textures.

Additional Textures to Add Variety

Now that you've created the base set, you will find that no matter how well you created the texture, the repetitive nature of the tiles is difficult to escape. If memory allows, you can create variants for each of these base sets. You can also create unique "random" tiles. These are special-case tiles or groups of tiles that occur very sporadically (to minimize their recognition) but give the terrain a more natural look (see Figure 9). Try to create variants that can also be reused to maximize your texture usage, such as a set of three or four variants that all work together and allow you to mix and match them.

Once you have your entire set, you can always build another test screen that checks for any problems in the tiled layout. Without the use of multiple transition types, the terrain isn't as natural as it could be, but Figure 10 is a good example of a base set.



On the Horizon

The methods I've just covered are just one technique for terrain tile generation. Once you've mastered the basics, you may find that there are things you can do to enhance your textures even more. If your game requires the use of a graphics card, then you have some powerful animation possibilities at your disposal if you have access to the hardware texture calls. You will need to talk over the technical requirements with your graphics programmer, but you can easily achieve such effects as pulsing lava, flowing water, and moving steam with animated textures. The base requirement is that the game use a graphics card, but with the latest-generation games, this is quickly becoming an expected base system requirement.

todd growney, morgan roarty & laurent benes

Designing Arcade-Style Vehicle Physics

Simulation or arcade? This is the looming question that all developers of racing games face. A pure simulation model attempts to reproduce exactly what happens if you are driving a real vehicle. An arcade model is a fictional simulation of the driving experience. A pure simulation is desirable, because it tends to feel very realistic and gives the player a solid connection to the vehicle and the environment. But being realistic, it is also very difficult to control. Not many people can drive a car 100 mph around a tight corner and live to see the next turn. A pure simulation tends to control the game design. An arcade-style simulation is relatively easy to control but bears little resemblance to reality and is in danger of disconnecting the player from the action. Being that there are no constraints for an arcade simulation, most racing games use this model. The arcade model can be shaped to fit any game design. The best solution lies somewhere between pure simulation and arcade. In this article, we will identify the components of the vehicle simulation that should be real and those that should be arcade, based on our experience working on RUMBLE RACING for Playstation 2.

TODD GROWNEY | *Todd is an engineer at Electronic Arts. He has a degree in physics with a minor in mathematics. Todd specializes in physics and visual effects. While at EA, Todd has also worked on NASCAR RUMBLE, SUPERCROSS, and FUTURECOP: LAPD. Todd can be reached at tgrowney@pacbell.net.*

MORGAN ROARTY | *Morgan is a four-year veteran at Electronic Arts. Prior to RUMBLE, he has also worked on the EA Sports NASCAR and ANDRETTI RACING franchises. Morgan can be reached at mroarty@ea.com.*

LAURENT BENES | *Laurent published his first game in France at the age of 17. For the last eight years he has worked at Electronic Arts as a lead programmer for projects on 3DO, PSX, and Playstation 2. He has worked on several games, including SHOCKWAVE 1 & 2, FUTURECOP: LAPD, and RUMBLE RACING.*

for Playstation 2
Rumble Racing

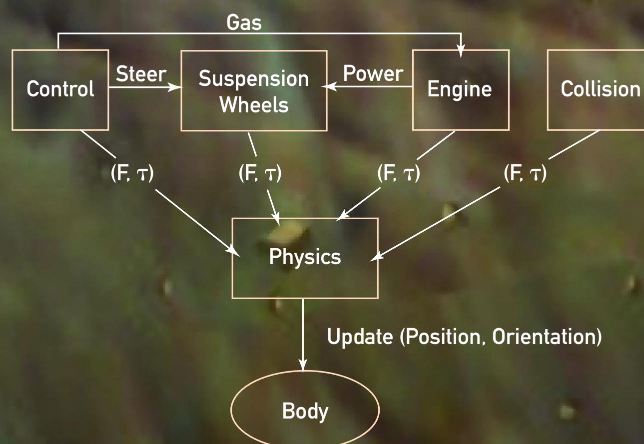
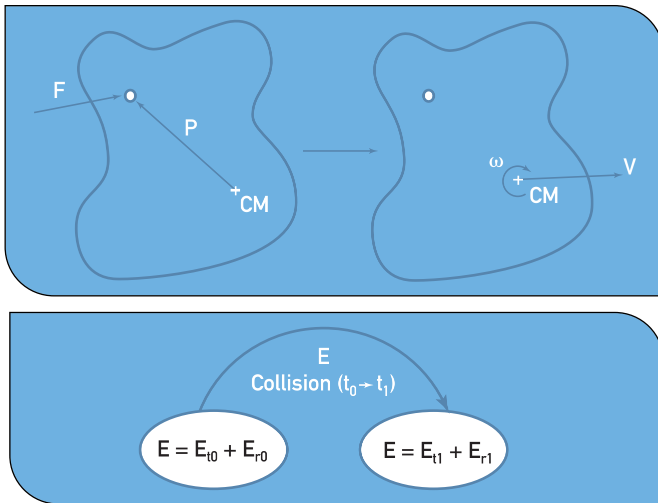


FIGURE 1. The major components of vehicle simulation and their interaction.



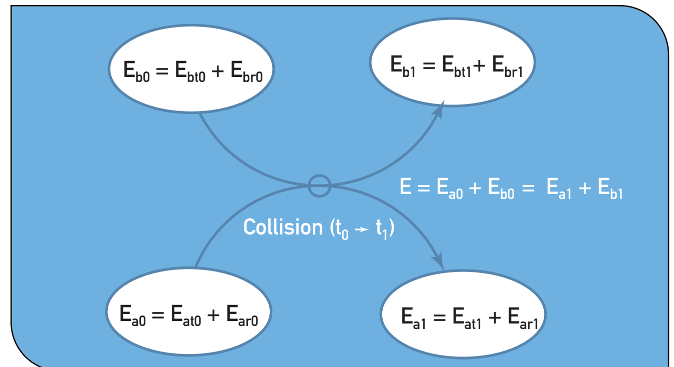


FIGURE 2 (top left). Force at a point P acting on a body changes V and ω. FIGURE 3 (bottom left). Conservation of energy for single body colliding with ground. FIGURE 4 (above). Conservation of energy for two bodies colliding.

Classification

The solution begins as it does for all software design, with classification. What are the major components of our vehicle simulation and how do they interact? A vehicle simulation can be broken up into five major modules: physics, collision, suspension, engine, and control (see Figure 1).

Physics. This module collects the forces applied at various points for a single frame of simulation and from these forces updates the vehicle's linear and angular velocities. These velocities are then used to update the position and orientation of the vehicle.

Collision. This module detects collisions of the vehicle body with the ground, walls, and objects. Once a collision is detected, the proper impulse force is applied to the physics module. These impulse forces are critical to both how real the simulation feels and how fun it is to play.

Suspension/wheels. The suspension module computes all forces generated by the suspension and the interaction of the tires with the ground. This is the trickiest module to write because the sources of force are numerous and complex.

Engine. The engine module simulates engine RPM, the gearbox, and power delivered to the wheels. This module also needs to simulate the feedback from the tires.

Control. This module turns the wheels and creates forces to be applied to the physics module in order to control the vehicle. In its most basic form this module has one simple task: Connect the player with the game. This module can single-handedly make or break your game.

Physics

The physics module should be as close to reality as possible, regardless of whether you are writing a simulation or arcade-style game. The types of forces applied to the physics module will determine the style of the game and how the player perceives the action. How the module changes the internal states of the body according to these forces should use real-world physics calculations. In other words, a force F applied at a position P on the body produces a change in velocity dV and angular velocity $d\omega$. This is the only task the physics module has, but as we will see, it is not a simple one.

A force F applied at a point P on a body of mass M and moment of inertia I will cause the velocity V and angular velocity ω to change (see Figure 2). But how much does each change? Like most physics problems, it is easier to grasp if you think of it in terms of energy. Applying a force F over a distance d to a body increases the kinetic energy of the body by an amount equal to $E = Fd$. This energy will be split between rotational and translational energy such that their sum is equal to E. Translational energy can be stated as:

$$E_t = \frac{1}{2} M V^2$$

And rotational energy can be stated as:

$$E_t = \frac{1}{2} I \omega^2$$

The trick now lies in determining how the energy is split between the two. This problem does have an exact solution, but an approximation can be used that will yield a result that is very real. First, there are two laws you need to know. The law of linear motion states:

$$F = M\left(\frac{dV}{dt}\right)$$

where F = force, M = mass, V = velocity, and t = time.

And the law of angular motion states:

$$T = I\left(\frac{d\omega}{dt}\right)$$

where T = torque, I = moment of inertia (tensor), and ω = angular velocity.

What is the inertia tensor? In our simplified universe, it is a 3×3 matrix describing the distribution of mass in the object being torqued. It is important to note that for a box of even mass distribution, I is a diagonal matrix:

$$\begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix}$$

This gives three simple independent equations we can use to solve the angular velocity change about each axis. Note that for this to be true, *T* must be in the reference frame of the box.

$$T_x = I_x\left(\frac{d\omega_x}{dt}\right) \text{ or } d\omega_x = \left(\frac{T_x}{I_x}\right) * dt$$

$$T_y = I_y\left(\frac{d\omega_y}{dt}\right) \text{ or } d\omega_y = \left(\frac{T_y}{I_y}\right) * dt$$

$$T_z = I_z\left(\frac{d\omega_z}{dt}\right) \text{ or } d\omega_z = \left(\frac{T_z}{I_z}\right) * dt$$

At this point it's important for us to keep in mind the law of conservation of energy, which states that the total energy of our system is maintained over time. From this we know that the sum of translational and rotational energy is equivalent before and after a collision (from timestep 0 to timestep 1).

$$E_{t0} + E_{r0} = E_{t1} + E_{r1}$$

$$E_t = \frac{1}{2}MV^2$$

where E_t = translation energy, *M* = mass, and *V* = velocity.

$$E_r = \frac{1}{2}I\omega^2$$

where E_r = rotation energy, *I* = moment of inertia, and ω = angular velocity.

Before a collision the total energy is known:

$$E = E_{t0} + E_{r0} = \left(\frac{1}{2}MV_0^2\right) + \left(\frac{1}{2}I\omega_0^2\right)$$

where V_0 = initial velocity and ω_0 = initial angular velocity.

After the collision, *E* is distributed between E_{t1} and E_{r1}.

$$E = E_{t1} + E_{r1} = \left(\frac{1}{2}MV_1^2\right) + \left(\frac{1}{2}I\omega_1^2\right)$$

where V_1 = final velocity and ω_1 = final angular velocity.

If you know V_1, you can calculate ω_1 and vice versa. In order to solve for these two exactly, one needs the particulars of the collision. If you can approximate one variable, you can find the other such that energy is conserved (see Figure 3).

In the real world, energy is lost in the collision in the form of heat or damage (to both the ground and the object). This would change the equation to $E = E_{t1} + E_{r1} + E_k$ where E_k is the lost energy.

The problem is the same when two moving objects collide; however, the energy can now spread itself out between four destinations E_{at0}, E_{ar0}, E_{bt0}, and E_{br0} (see Figure 4). Now you need three variables in order to find the fourth. Also, to find the exact solution you need three more independent equations describing the collision. A very long and potentially unpleasant road begins here, unless you're into that sort of thing.

Just as before, if energy was lost to heat or damage, the equation would become $E = E_{a0} + E_{b0} + E_{a1} + E_{b1} + E_k$ where E_k is the lost energy.

The behavior of a box colliding with the ground or two boxes colliding with each other does not have an exact solution (like all physics problems based in the real world). All you can do is make a best guess at the solution using basic laws such as the conservation of energy to help guide you. For example: A box collides with the ground rotating at some linear and angular velocity. After the collision, it will have a new linear and angular velocity. We can calculate the energy (E_0) before the collision. Let's assume 20 percent of the energy was lost, or $E_k = 0.2 * E_0$. This leaves us with:

$$E_0 = \left(\frac{1}{2}MV^2\right) + \left(\frac{1}{2}I\omega^2\right) + 0.2 * E_0$$

or

$$0.8 * E_0 = \left(\frac{1}{2}MV^2\right) + \left(\frac{1}{2}I\omega^2\right)$$

If we estimate ω we can determine exactly what *V* should be (and vice versa). Note that this equation is a scalar equation, so it only tells us about the magnitudes and not the directions.

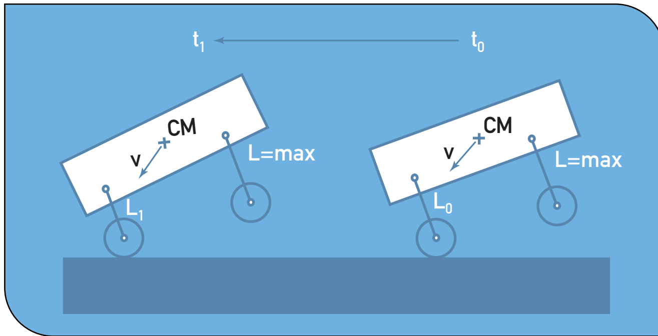
Collision

This module has the fairly straightforward task of detecting collisions with the ground, walls, and other objects. In RUMBLE RACING, the cars were physically seen as boxes as far as the collision was concerned. The world borders (or fences), for simplicity, were seen as vertical fences. Cylinders, spheres, and lists of bounding boxes were also used to represent other objects.

Collision detection was handled in two ways: object to object, and object to world. For the object-to-object collision detection, every object was top-down positioned in a circular array (position along the track gives the position in the array). The caveat with this approach is that one object on the main route and one on a shortcut could be positioned in the same cell. Each object in a cell was tested against other objects in the same cell or adjacent cells. Each object could be an active collider or regular collider. For a pair of objects to be tested for collision, at least one object had to be an active collider. This cuts down on the pairs of colliders quite a bit; who cares if two pieces of an exploding chair collide with each other? Note that this circular array was later also used as a proximity array for the car AI, to determine quickly which objects in their surroundings they might want to pick up or avoid. When you are dealing with hundreds of objects and want 60 frames per second, every saved cycle is a good cycle.

For a given pair of objects being tested for collision, the collision system conducts a quick bounding-sphere test. If positive, a more complex box-box test is done. For object-to-world collisions, a custom box-lozenge test is performed.

Once a collision is detected, the position, direction, and magnitude of the impulse force resulting from the collision must be determined. In short, we must compute the *F* and *P* from Figure 2 to be sent to the physics module. This impulse force is central to a realistic collision model. Consider the following simple example. Suppose a bar were to fall onto a flat surface. At impact its velocity is *V* and it is not rotating (zero angular velocity). Its mass is *M* and moment of inertia is *I*. After the impact, what will its velocity and angular velocity be? If the physics module is accurate, all we need is the position of impact and the force created there. It



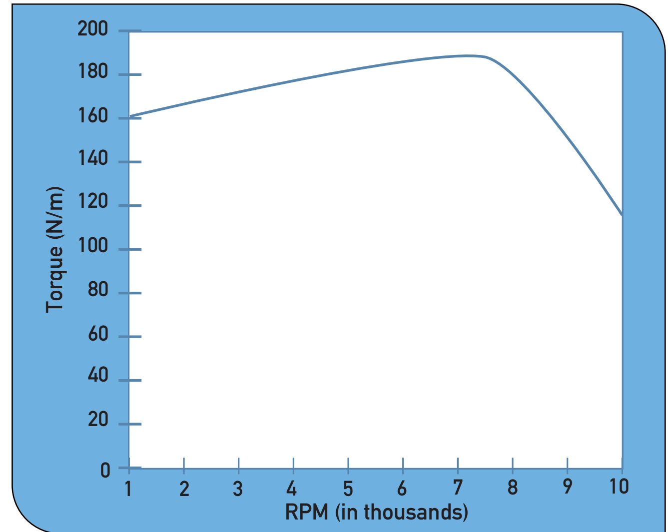


FIGURE 5 (above). Suspension compressed as a result of vehicle motion.
FIGURE 6 (right). Example torque curve.

quickly becomes clear that the calculation of this force and what the physics module does with this force are interconnected.

An alternative approach that is much more conventional and straightforward is to update the linear and angular velocities directly (making sure you conserve energy). Once you have done this you can estimate the impulse force that would have been required to cause this change. If your simulation is simple enough, you may not even care about this force and therefore not compute it.

To detect collisions with the ground, we maintained the corner positions of the object's bounding box for each frame. If the altitude (Y-component) of a corner is smaller than the altitude of the ground at that point, then the corner collided with the ground. If the system finds that one or more corners have collided, it averages those points together to get a close approximation of the point of collision. Then the entire object is displaced out of the ground by the depth of the corner with the deepest penetration. After that we compute new linear and angular velocities, or we can compute the impulse force and apply it to the approximated intersection point. Choose your poison. We're still not sure which method is better.

One of the key demands of our collision system was to have very fast access to world data so that elevation and normal vectors at a given point could be retrieved. Given the Playstation 2's small cache and slow random-access memory, we spent quite a bit of time ensuring that we wouldn't have to traverse a huge database each time a query was to be done.

Suspension/Wheels

The suspension module should be as realistic as possible but modified slightly from reality to suit the game design. This is perhaps the most complex module of them all.

Suspension. As a result of vehicle momentum (both linear and angular) and changes in ground elevation, the suspension will become compressed (see Figure 5). This compression of the suspension results in forces being applied at the positions where the suspension is attached to the vehicle. In each simulation frame we first update the orientation and position of the vehicle body. Given the new position and orientation (including tire radius and any other geometric definitions), we then calculate how long each wheel's spring would have to be to place the tires on the ground. Of course, if a spring reaches its maximum length, the tire leaves the ground and does not influence the vehicle. With these new spring lengths we can now calculate the force of each spring on the vehicle as $F_s = k * L$, where k represents the strength of the spring and L is the length. The force created by the shocks (dampeners) is:

$$F_d = c * \left(\frac{dL}{dt}\right)$$

where dL is the change in length for this frame, dt is the time since the previous frame, and c is the shock strength.

Spring force:

$$F_s = k * L_1$$

Dampening force:

$$F_d = c * \left(\frac{dL}{dt}\right) = c * \frac{(L_1 - L_0)}{(t_1 - t_0)}$$

There is an extremely important force that is easily overlooked and that this model completely ignores. What happens when the shocks completely collapse? At this point the shocks do nothing, and the vehicle is bouncing off the tires and big rubber stoppers in the shocks. The forces generated from this collision are unbound and can get very large very quickly. We like to call this the bottom-out force. It is calculated much like a collision of the vehicle body with the ground. An accurate response to this force will cause violent rotations of the vehicle as the vehicle lands at an angle or as the front end bottoms out when the vehicle hits a steep jump.

Wheels (tire/ground interaction). The tires are ultimately what steer and accelerate the vehicle. When you apply gas to the engine, the engine applies torque to the tires. This torque results in a force applied to the vehicle in the direction the tires are accelerating (longitudinal). When you steer, the vehicle's tires are turned away from the direction the vehicle is traveling. The angle between these two directions is called the slip angle and results in a force applied to the vehicle perpendicular (lateral) to each tire. If

we project these longitudinal and lateral directions into the plane of the ground and use the normal of the ground, we form an orthogonal reference frame called a patch.

We project all forces including weight and spring force into each patch. It's important to note that the springs apply the same force to the tires as they do to the vehicle. Next, we project a fictitious force generated by cornering. We also add the force delivered by the engine to the wheels (in the longitudinal axis). This force is simply the delivered torque divided by the tire radius ($F = T/R$). After summing all these forces in the patch referential, it's time to see how much the ground is willing to push back. This is where static and kinetic friction play their parts. If we multiply the portion of the patch force that is normal to the ground by the static friction coefficient, we get the maximum force that the ground is willing to push back. If the patch force in the plane of the ground is less than this force, the ground pushes back with equal force. However, if this patch force exceeds the static force, the tire slips and now the ground only pushes back with a force equal to the kinetic force.

Tuning the Vehicle Physics

Physics Params

Parameter	Value
Spring Len	0.031250
Spring Force	2554.872559
Spring Dampening	0.424573
Tire Quality	1.911241
Side Slip Friction	0.311000
Rollover	0.604239
Min Steer Angle	0.075190
Min Steer Turn Rate	0.154427
Weight	2513.491211
Combat Mass	0.500000
Engine Power	1.000000
CM Fore/Aft	0.000000
CM Up/Down	0.000000
AirDrag	10.000000
suspensionDispFactor	0.750900

In-game tuning control panel.

Our top goal for the cars in Rumble Racing was to make sure that each car felt and drove differently. Secondarily what was essential was to make sure that the handling matched the appearance, that every car was capable of winning, and that the disparity of feel was wide enough so that everyone could find a car that they liked to race with.

It is important to note that the top speed for each car class (Rookie, Pro, and Elite) was the same for each car in the class. In other words, the Rookie Thor's (the rocket car's) top speed of 140 mph is the exact same as for our smallest-engine car, the Banger. This was done deliberately in order to keep the game interesting no matter what car you were driving so that you would drive them all. However, this provided a unique challenge to the tuning.

Method. We had a special version of the game that exposed all the tuning parameters via easily modifiable in-game menus and commands (see Figure). All the parameters were available for modification on the fly, giving us an immediate feedback loop. This was especially important because we could run a lap, make a tweak, and get immediate feedback on the change.

The modifications were then written out to text files for later recovery. These text files were handed off to our physics engineer, who incorporated them into the game. We maintained a spreadsheet containing all the cars and individual settings for cross-referencing, which became invaluable for examining the relative differences between values across the range of cars.

In addition to the car's parameters, additional in-game tuning panels allowed us to test and define the various terrain characteristics, such as roughness, resistance, and grip. Rumble Racing had around 20 different terrain types (pavements, dirt, mud, grass, and so on). These were fine-tuned continually as the tracks came together.

A quick example of how this benefited us is in the initial Sand settings. For the sand tracks like "Surf and Turf" and "Sun Burn" the resistance was set too high and the grip a little low, causing cars to get stuck in some spots. This also made the course too slow to race through. We firmed up the sand a bit in order to bring the gameplay up to where we wanted, and to make sure cars wouldn't get stuck.

Car tuning approach. Our initial approach was to get one car tuned and then vary all of them off this baseline. We tried to be very reasoned and analytical about the tuning, but in some cases the final settings came down to particular combinations among many different variables. It was a highly iterative process that took a lot of balancing and recalibration as we went along, but in the end it all worked.

We started with the most generic cars (such as the Cobalt and Serpent) first, and then moved onto the smaller compact cars, the heavier American cars, trucks and vans, and finally the exotic and special cars. We nailed the first generic car, the Cobalt, very quickly, which then became the initial baseline off which everything else was varied.

When working on a new car, the main areas we started with were the suspension, side-slip friction, steering angles, and steering rates. We usually drove a lot of laps trying many different values for these settings before we felt like we had something that worked well. Then we moved on to the engine power and car weight. After getting these values close we did a lot of fine tuning and tweaking, comparing all the variables before finalizing a setup.

Car class modifications. Initially the goal for the compact cars (such as Tiberius and Maelstrom) was to have a quick, well-handling feel. The notable changes that made this happen were modifying the suspension settings, improving the tire quality, and decreasing side-slip friction. To keep these cars in balance with the rest of the game, the engine power was reduced by 25 percent from the generic cars.

It took a while, but once these cars felt good we locked in those settings. These settings were so well received and popular that we resisted changing them much after we locked them in. We really became hesitant, and almost fearful, about any modification to these cars. In some sense the baseline shifted to this set of cars.

The heavy American-style cars were the most challenging to tune. The goal was to have them heavy-feeling and not as precise to control, but with a very powerful engine. We really struggled with these settings but discovered through a combination of lowering the angle and rate of steering we could greatly affect the feel of these cars. It was the key to their tuning and also an important feel modifier we used later on some of the unique cars.

On the trucks, one afternoon we started experimenting with their ride height (suspensionDispFactor). At the time the trucks were all 4x4 off-road style with a bouncy suspension. During the process of experimenting with the ride height we tried lowering some of the trucks down to look like a low rider, which turned out well. We kept the physics the same on the 4x4 versions, but the dropped truck versions had to have their physics tweaked to match the handling to the new appearance. It quickly provided some variety.

Lay the groundwork. Tuning the cars for Rumble Racing was an interesting process that happened over a three-month period. The physics engine was basically locked at two months before alpha, but the individual car tweaking and tuning lasted well into beta. It was extremely helpful that we had a good sampling of representative tracks in place early so we could get an early determination of the state of the physics. We eventually picked out three of our most varied tracks and used them as the test bed for any physics engine changes and tuning confirmations.

Overall, tuning the cars was a production dream. The physics was done early and was extremely adaptable to all the varied cars and types of tracks. Car tuning is a lot of hard work and long hours, trying out many different variations of the settings and doing laps over and over again.

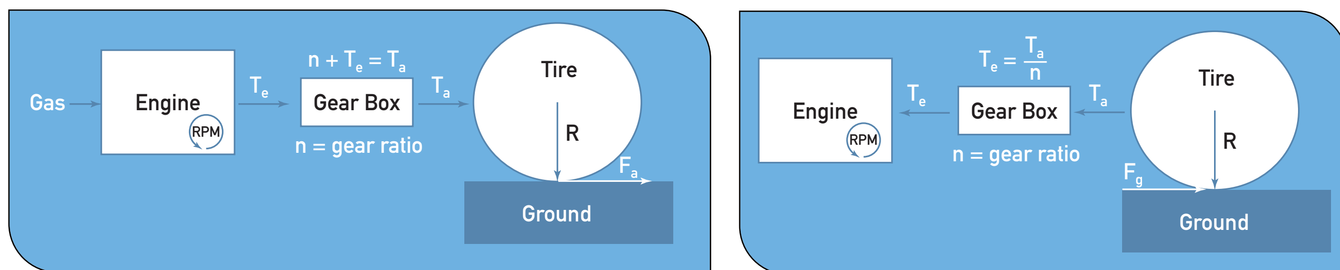


FIGURE 7 (left). Engine pushes on ground. FIGURE 8 (right). Ground pushes on engine.

Engine

The engine is supplied with gas which in turn accelerates it, increasing its RPM. As a result, the engine produces a torque which is dependent on the current RPM. This RPM-to-torque relationship is known as the torque curve (see Figure 6). It is important to note that the engine creates a torque, which is then multiplied through the gearbox. This allows a different torque to be delivered to the wheels depending on which gear the vehicle is in. This delivered torque pushes the vehicle forward (see Figures 7 and 8).

It is also important to note that to get a decent simulation you also must consider the reverse effect. We call this effect the feedback loop. Imagine the vehicle is in gear but you are giving it no gas. If the vehicle is on a hill, the force of gravity will accelerate the vehicle. In this case the ground is delivering a torque to the wheels which is then divided back through the gearbox and applied to the engine, thus accelerating it. This effect is also evident when you are moving too fast for a given gear. The engine revs way up and the vehicle rolls forward on its suspension because of the resultant torque on the tires produced by F_g at R.

Control

This is the module that mostly defines the style of game; it is the cornerstone of the style, if you will. Obviously its main purpose is to steer, accelerate, and brake the vehicle. You can think of the other modules as the control system's toolbox. If the tools are crap then the result will be crap, no matter how hard you hack. Let's consider a laundry list of responsibilities for the control system.

Steering. By interpreting the user inputs, we turn the wheels. By turning the wheels, we change the slip angle of the tires and thus affect the cornering of the vehicle. In a pure simulation you'd be done and the suspension module would take over from here. But what if we don't like the result of the simulation? It is now my pleasure to introduce the most powerful equation in all racing games, the steering equation:

$$\omega_y = \frac{(V_z * S)}{D}$$

In its simplest form it states how fast to rotate a vehicle ω_y depending on its speed V_z, wheel base D, and steer angle S. This equation assumes much about the conditions of the simulation and thus becomes restrictive if used as the sole source of steering. However, by scaling between the pure simulation and the steering equation, you can dynamically control where you lie on that line between control and chaos. So how do we scale between these two extremes? It's quite simple. First, we calculate the angular velocity the steering equation thinks the vehicle should have. If we take the difference between the steering equation and the actual result based on the suspension module, we can determine the torque that would have to be applied in order to conform to the steering equation's result. This gives us our scale. If we want to behave exactly as the steering equation suggests, then we apply all of this "balance" torque. On the other extreme, if we want to challenge the player or do nasty things to the vehicle, we can just ignore the balance torque and let the chips (or chunks) fall where they may. At low speeds you want to let the natural physics do their thing, thus allowing crazy burnouts. As you progress to higher speeds you want to scale back the real simulation and increase the more predictable steering equation. This gives the player more arcade control in the realm of the insane. Let's face it, 220 mph on a back country road is crazy!

How the vehicle responds to the player steering the vehicle is a very delicate problem. At low speeds you want to be able to turn sharp and fast. At high speeds you want to turn smaller angles and with a slower response. By scaling the maximum steer angle according to speed, with the largest at rest and smallest at maximum speed, you gradually narrow the maximum steer angle as you go faster. Remember that the resulting turn is the product of vehicle speed and steer angle, so this won't restrict your ability to turn at high speeds.

The steer speed, however, is not an absolute — it should be dependent on the current steer angle, slowest at wheel center, and fastest at the maximum steer angle. Also, if the player is steering opposite the current steer angle, you want to double or even use the maximum steer speed. The maximum steer angle, steer speeds, and all other controls should be tuned so that the vehicle glides through the average turn at the average speed.

Analog steering requires some additional modifications. First, the actual analog steer amount should be made nonlinear so that there is little change in the center of the control and more at the extremes. This desired nonlinear behavior can be achieved by squaring or cubing the analog value received from the controls (assuming the analog values range between –1 and 1, and you maintain the sign). For steer speed control, imagine the analog steer angle being tied to the actual steer angle by a spring. How tight the spring is determines how responsive the steering is. The same maximum steer angles that apply to analog also apply to digital controls.

Acceleration/braking. Let's say after tuning the simulation and tire quality and achieving a decent balance you want to increase

acceleration and reduce braking time. If you simply increase tire quality out of the natural realm, you will begin to flip the vehicle when accelerating, braking, and cornering. Our solution is to leave the tires alone and apply a fictitious force to the center of mass to get the desired acceleration. You can do the same thing for braking.

Burnout. Everybody loves a good burnout. So if the vehicle is going relatively slowly and the player punches the gas, you can start a burnout timer. This timer continues to reset as long as the player holds the gas and cranks the wheel. The value of this timer is used to reduce the rear tire quality to such an extent that wild, smoky burnouts can go on for days. But as soon as the player shows the slightest indication of wanting to get the hell out of Dodge you allow the burnout timer to expire, quickly ramping up the rear tire quality to normal. This sudden ramping of friction causes the front end to torque up slightly — joy!

Slide control. In a real simulation, if you try to turn a tight corner at 200 mph you will most certainly go straight into a wall or roll the vehicle. To avoid this, we introduce two handy controls we like to call extra side-slip friction and roll control.

The first control, extra side-slip friction, works as follows. If we project the vehicle's sideways velocity onto the ground and apply a friction to that velocity we have a fictitious force we can apply to the vehicle:

$$F_s = -s_{sc} * V_s$$

where V_s is the projected side velocity and s_{sc} is the side-slip coefficient. The greater s_{sc} is, the less the vehicle slides out around corners.

One of the best sources of feedback in a racing game is the rolling and pitching of the vehicle on its suspension as it corners and accelerates. This rolling can quickly become rollover if nothing is done. By isolating the lateral forces produced by the tires we can control the rollover. Again, a force at a point on the vehicle can be split into a force on the center of mass and a torque. If we apply the force at the center of mass and scale the torque, we can control the rollover without affecting the general momentum of the vehicle. Pick some maximum rollover angle and scale the torque between 0.0 at or above the maximum angle and 1.0 at the zero angle. This in effect smoothly cuts off the rollover torque as you near the maximum angle while allowing a wild ride in the center.

Balancing control and chaos. Chaos is fun to watch but is a form of punishment for the player. A delicate balance must be maintained so as not to frustrate the player. For example:

- If you drive a vehicle straight off a cliff, the rear suspension pushes up on the rear of the vehicle when the front tires go off the cliff and are hanging in midair. This unbalanced force from the rear suspension just as you go over will roll the car forward. If you are airborne long enough, the vehicle will flip over.
- If you turn a corner too sharply and your center of mass is too high, you could very easily roll the vehicle.
- If your trunk explodes (for example, your last shipment of moonshine ignites), your vehicle would be tossed into the air and you could land on your side or upside down.

If the player did nothing wrong, you don't want to allow this kind of chaos to take over. At the same time, if it's time to punish the player you need to get him or her back in the game quickly after doing so. If you make the punishment look good enough, players will tolerate it for a short time, especially if they are hurtling in the general direction of their goal.

This is where the cat reflex comes in. The cat reflex has one purpose, to eliminate player frustration due to chaos (flipping, rolling, and other dramatic effects). But beware, if the cat reflex does its job too well, it can mask or completely overwhelm the pure simulation and the game will look stale (which is what we are trying to avoid). To maintain stability while not seeming rigid, we define a maximum angle of rotation that the vehicle can have. If the vehicle rolls or pitches more than this angle, we dampen the angular velocity to halt its rotation out of this "stable" region. By doing so we allow natural rotation within this region, clamping it at an extreme. This maintains stability while allowing the vehicle to land in an infinite number of ways. There is nothing more boring than a vehicle that always lands flat on all four tires. If, however, we choose to punish the player because he or she ran over a bomb or got hit really hard, we disable the cat reflex. At this time we can toss players like an old shoe and wait until they begin sliding on the ground. When they're back on the ground we turn on a special version of the cat reflex which attempts to roll the vehicle onto its tires in the direction it is already rolling so it appears natural.

Closing Remarks

The player is much more sensitive to rotational motion than to translational motion. By affecting motion with only forces and torques you can have hundreds of seemingly complex influences which sum into just two vectors (force and torque). At the end of each frame, this net influence is used to update the linear and angular velocity, which is in turn used to update position and rotation. This technique is clean, simple, and powerful.

All physics equations in any book on the planet use MKS units (meters, kilograms, seconds). After updating the vehicle position in each frame, you have to convert into game units if you use meters in your computations. This is why we prefer to use GUKS (game units, kilograms, seconds). This requires no conversion, and distance (as well as velocity) is always available in the native game format. All this requires is that you write a set of macros to convert distance, speed, acceleration, force, and so on. These macros are then used to initialize your data. By spending time setting up this system and building a nice set of macros (and testing them carefully) you can avoid a lot of confusion.

Unpredictability is a good thing. By simulating multiple sources of opposing influences you can get unpredictable results. This feeling of unpredictability adds excitement and reduces any feeling of repetition. All you need to do is tune these results back to that fine line between control and chaos.

If something doesn't seem right in the simulation, nine out of ten times it's a simple bug in the code and not in the theory. All too often we have agonized over the correctness of a theory only to find out that we forgot a sign while typing in the equations. Check your code first, and then check the theory.

Simulation or not, it's a game. If it isn't fun, figure out why and fix it. There is a very fine line between fun and boring. One small modification can change everything instantly.

The (Not So) Dark Art of Scripting for Artists

Whether it's MEL, MaxScript, Perl, Python, or even JavaScript, most modern 3D applications give users access to an internal scripting language which enables them to accomplish myriad tasks. You may wonder why, as an artist, you should invest the time to learn some obscure collection of ifs, fors, and thens. The answer is simple: Scripting makes your life easier. Whether it's automating a repetitious task, doing something the computer is better at than a human (such as creating anything random), or adding new functionality that your program lacks, a clever artist can accomplish all this, and more, with scripting. While scripting has limits to what it can achieve, long gone are the days when adding functionality required a tools programmer to create a custom plug-in. In this article, I will take you through the basics of scripting and why it's useful, illustrating this through the construction of three projects that highlight reasons you may want to explore scripting.

First, let's start with some basics. One important thing to remember is that while I'm going to use the Maya Embedded Language (MEL), the scripting language for Alias|Wavefront's Maya, almost all the concepts are similar between art packages. Generally, each 3D art package supports its own scripting language and perhaps a few standard languages (see sidebars).

At its most basic level, a script is a bunch of commands. Instead of pressing buttons or moving objects manually, the computer does it for you. Most programs have a window that will echo every command you perform. For example, when you simply press the shelf button to create a polygonal cube, Maya's Script Editor (its way of inputting and echoing commands) shows something like this:

```
polyCube -sx 1 -sy 1 -sz 1 -h 10 -w 10 -d 10 -ch 1;
```

This translates to: "Create a 10×10×10 polygonal cube that has one subdivision in each axis and has construction history." Because Maya allows for shorthand notation in its command structure, reading the scripts it produces can often be confusing. The preceding command could just as easily have been written as follows:

```
polyCube -subdivisionsX 1 -subdivisionsY 1 -subdivisionsZ 1 -height 10 -width 10 -depth 10
   -constructionHistory true;
```

As you can see, this is considerably easier to read, but actually writing it out can take a while. Sure, that one command doesn't seem like much, but multiply it by 300 or 400 lines, and you can spend all morning writing what could have been accomplished in 20 minutes. Many of the flags (the "options") are similar across commands, so the shorthand can be learned quickly, and you will soon be able to read the abbreviated versions as easily as the long versions.

Well, big deal. So you can make a cube. The important thing to realize here is that it wasn't really pressing the button that created the cube, but rather the button issued a MEL command that did it. Keep the Script Editor open as you work, and you will see that everything you do issues a MEL command. Our first project uses the simplicity of these echoed commands to build your script for you.

DAVID STRIPINIS | *David is currently director of animation for Factor 5. He would like to thank all the people who held his hand while he learned to script. And if a programmer comes after you with a blunt object, he advises you to chase them with a pointy one. He can be reached at david@factor5.com.*

Illustration by Boris Kulikov

MaxScript Magic

MaxScript is the interpreted scripting language used to control Discreet's 3DS Max. Code written in MaxScript can be entered and executed using the interactive Listener console. The Listener provides a traditional command-line interface (CLI) which allows access to the bowels of Max without using the GUI. To experiment with MaxScript, launch Max and then press F11 to open up the Listener console. The Listener interface is divided into two parts: an upper "MacroRecorder" section and a lower "Console" section. When the MacroRecorder feature of the Listener is enabled, commands that are received from the standard UI will have their MaxScript equivalent echoed to the MacroRecorder. While the MacroRecorder can be a useful tool for discovering how Max itself accomplishes tasks, we will stick to the basics and try entering a few simple commands of our own directly into the Listener's Console section.

Once the Listener is active, enter the following command into the console:

```
box pos:[0,0,0]
```

Keep your cursor located on the line you have just typed and press the Enter key on the numeric keypad to execute the line. If you have everything entered correctly, a new box will be created at the center of the scene, and the Listener will reply:

```
$Box:Box01 @ [0.000000,0.000000,0.000000]
```

Whenever you press the Enter key with the Listener active, it causes Max to interpret the content of the line that the cursor is currently located on as a script command. The entire content of the Listener console can be edited and then re-evaluated using standard text-editing commands. Use the cursor keys to move back up to the box command and edit it in place to show:

```
box pos:[0,0,100]
```

Press Enter again to cause the Listener to reinterpret the changed line. This will create another box located 100 units away from the first (along the Z-axis). The Listener will respond with:

```
$Box:Box02 @ [0.000000,0.000000,100.000000]
```

Testing different variations on commands is just that quick. There is no need to exit Max, edit your source, compile the changes, relink, restart Max, and finally test the revised executable. The immediacy of the Listener interface makes it easy to experiment.

Each of the following one-line commands is a simple variation of the `for` loop control. They use the `random` function to generate values between two range parameters. To add 100 boxes, scattered randomly:

```
for obj in 1 to 100 do box pos:(random [-100,-100,-100] [100,100,100])
```

Select some of the boxes that you have just created, and you can move them randomly using:

```
for obj in selection do move obj (random [-100,-100,-100][100,100,100])
```

Finally, this command will rotate all selected objects around their local Z-axis:

```
for obj in selection do rotate obj (random (eulerangles 0 0 0)
   (eulerangles 0 0 360))
```

Another useful capability of MaxScript is the ability to bind code to menu buttons. Instead of pressing Enter after code to execute it, select your code in the Listener with the mouse and then drag and drop it onto the Max tool bar. This will create a custom button that you can press to execute code automatically. Read the online help on MacroScripts for more information on this feature.

If you are interested in exploring MaxScript in more depth, the book *Mastering MaxScript and the SDK for 3D Studio Max*, by Alexander Bicalho and Simon Feltman (Sybex, 2000), is an excellent resource.

— Mike Biddlecombe, Gas Powered Games

Project 1: Repetition, Repetition, Repetition

The life of a game artist is often filled with repetitive tasks. Who hasn't groaned in frustration when a programmer comes to you asking if you can change something about an art asset? To the programmer, it could be a simple request such as renaming a few objects because the new engine revision can't handle underscores in the names of objects. To you, it's a wasted morning, as you change the 145 files that include the object and re-export them. Your time could be much better spent tweaking a cutscene, or better yet standing around the snack machine discussing the intricacies of COUNTER-STRIKE tactics with your coworkers. Sure, you can go into each file, find the appropriate objects, rename them, save the fixes, and export the new scene. But this is both a colossal waste of time and also prone to error. If you make a typo, that programmer will come back, possibly with a large, blunt object, looking for you to do the whole thing over again. Wouldn't a way of doing this automatically make your life much easier? If, after doing the process once, you take a look at the Script Editor, you'll see something like this:

```
select -r engine_detail_01;
rename "engine_detail_01" "engineDetail01";
// Result: engineDetail01 //
select -r engine_detail_02;
rename "engine_detail_02" "engineDetail02";
// Result: engineDetail02 //
select -r engine_detail_03;
rename "engine_detail_03" "engineDetail03";
// Result: engineDetail03 //
```

If you look at what you've done, it's just to select and rename three objects. Those lines prefaced with "//" are called comments. Comments are a very special and useful part of scripting. They are lines of the script which the computer ignores, and which provide someone reading the script with an idea of what the script does. Maya also issues feedback in the Script Editor via comments, so you can cut and paste sections of the echoed text without causing errors. If you now save the scene, you will see Maya echo:

```
file -save;
```

Now we need to export the scene, but we need a "generic" export feature; otherwise, we will always export to the same file. Luckily, it's given to us by Maya. By pressing Control and Shift and clicking on Export Scene, we can create a button that executes the command "`ExportScene`." We can see this simply by dragging the new shelf button to the Script Editor. If we investigate further, we find the generic "`OpenScene`" command as well. Let's put our script together with just a little copying and pasting:

```
rename "engine_detail_01" "engineDetail01";
rename "engine_detail_02" "engineDetail02";
rename "engine_detail_03" "engineDetail03";
file -save;
ExportAll;
OpenScene;
```

We removed the feedback comments, which are unnecessary, as well as the commands that selected the objects, because you don't need to select an object to rename it (it's simply a by-product of working by hand). When executed, the script renames the three objects, saves the scene, and opens a file requestor for you to export the scene. As soon as that is completed, it opens a requestor for the next file to open. While a more complex script could automatically export and open the next file, we are trying to keep it simple right now. In Maya, we can drag and drop this script to the shelf. So in order to make our script perform the modifications, we simply have to press a button. Now we can do all those modifications and exports in a minimal amount of time and without error. You have also written your first script.

Now, you may feel rather unfulfilled at this being called a script. You probably think of scripts as things that produce drastic effects, have immense custom windows, and so on. While those types of scripts can be created, the power of scripting for the artist is in accomplishing mundane tasks. Keep in mind that with many scripts, simplicity is the key to success.

Project 2: Starting to Get Fancy

Our next step is to create a command of our own, similar to the `ExportScene` from the previous example. The reason for doing this is simple: You don't want to rewrite a script every time you need to perform some common function you need. So let's create a command that we can execute from either a button or the Script Editor. In Maya, there is a concept called a procedure. A procedure is a collection of MEL commands to be executed in order. For our second project, we will create a procedure that will use the unique functionality of the computer to our advantage. Computers do a few things really well, such as math. Humans are very good at picking out randomness but very bad at creating it. The computer, however, is very, very good at creating randomness. As such, most languages have a function for creating random numbers, and Maya is no exception. Let's create a procedure to move all selected objects around randomly. First, you'll need to understand the concepts of variables and arrays.

Variables. A variable is simply a placeholder. It can be a number, a word, a whole phrase, or even something esoteric, such as a vector. A variable allows you to create a generic command that can have certain parts of it replaced, or to do math functions based on conditions that were unknown when the script was written. I know that for many artists math is a scary thing, but it shouldn't be. The mathematics involved in scripting is usually very simple, and anyone who can understand a Cartesian coordinate space should have no problem. Variables in MEL are signified with a leading `$`. So if you create a floating-point variable called `$myNumber`, it can hold any number imaginable, including fractional numbers. In programming there are four basic variable types: integers, which are whole numbers; floating-point values, which can hold numbers with fractions; vectors, which contain three floating-point components; and strings, which contain text.

Arrays. An array is simply a collection of variables of the same type. Say you want to store the names of all the objects you have in the scene. Rather than having a series of string variables called `$myObject01`, `$myObject02`, and so on, it is much more efficient and useful to create one variable called `$myObjects`, which can hold all of them. In Maya, this is done by appending "`[]`" to the variable declaration at the beginning of the script. While this may seem bizarre, these weird annotations and symbols actually come in handy.

So let's get back to writing the procedure. First, create a file called randomMove.mel. We want to put a comment header at the top of our script, indicating what the script does and how to use it. Other useful things to put in the header are your name, the date of creation, and what version of software the script was created for. Make sure you put the comment marker "//" at the beginning of every line. We don't want Maya trying to execute your e-mail address.

```
// randomMove.mel
// Randomly moves selected objects up to 100 units in each direction.
// Author: David Stripinis E-mail: david@factor5.com
// For Maya 3.0
```

Next, we begin the actual script:

```
global proc randomMove ()
{
```

These lines indicate to Maya that we're creating a command called `randomMove`. From now on, whenever we type "`randomMove`" at the command line, it will execute the procedure between the curly brackets. Bracketing is a very important aspect of scripting, as with all programming. It tells the computer, "These parts stay together." Within a script, you may have subsections that need to be completed before moving on, and bracketing is also used there. Let's continue to the variable declarations:

Scripting in Softimage

The latest version of Avid's Softimage XSI, version 1.5, supports a variety of scripting languages. For simple scripts, XSI is very similar to Maya or 3DS Max: if you open up the script editor window, you can see all the commands that are being executed as you create and modify objects. You can also type new commands into the script editor window to execute them, or drag commands out of the window and tie them to new menu items in the interface.

If you're creating more complicated scripts, you can take advantage of XSI's support for ActiveScripting, which enables the use of many different scripting languages. Using VBScript, JavaScript, Perl, or Python, you can execute any action that generates a command in the log window (generally user interface actions, importing, and exporting). In addition, XSI can also communicate with other programs that support ActiveScripting. For example, you could write a script that launches your e-mail program and sends you a message at home when a batch process has completed.

The Softimage tutorials on scripting are very thorough, and the online command reference details the actions you can perform via the script interface.

— Mark DeLoura

```
// declaration of variables
string $selectedObjects[];
float $randomX, $randomY, $randomZ;
```

These variables will allow us to hold things that may change each time we run the script. They need to be declared so that Maya knows what type of value each variable holds.

Next we need to tell our script the values of some of these variables. Our first task is to get the list of all currently selected objects; we will use the command `ls`, which stands for list. Unfortunately, "list" is not a command, so this isn't as intuitive as you might like. The way `ls` works is simple: You ask for a list of items, using flags to filter the list so that you only get a list of exactly what you need. In our case, we want to get all of the objects that are selected. For this, we use the `-selection` flag, which can be abbreviated to `-sl`.

```
// build selection list
$selectedObjects = `ls -sl`;
```

The backward single quotes are similar to brackets, in that they indicate to the computer that it should execute the command in the quotes before doing anything else on that line. In this case, we assign the list of all selected objects in the scene to our previously declared string array.

Now we need to put in a little error checking. It's always good practice to put error checking into your scripts so that if there is a problem with the way the user is running your script, you can let them know exactly what is wrong. For example, receiving an error indicating that you can only select polygonal objects is a lot clearer than "Illegal operation in script myScript.mel line 82.7" The way we error-check is with what are known as conditional statements. Conditional statements require a criterion to be satisfied. Think of it as the computer's own little bouncer. Just like when you go out to a bar and they check your ID to make sure you are old enough, the program wants to make sure you satisfy its own rules before allowing you to continue through the procedure.

```
// check to make sure they did select something
if ( `size $selectedObjects` == 0 )
    error "Please select at least one object.";
```

This command makes sure that whoever is using the script has selected an object; otherwise, the procedure does nothing. The user might have thought it would automatically move all the objects in the scene. The "`size`" command counts the number of objects in our `$selectedObjects` array, and an error is printed if that number is exactly 0.

The final part of our script requires a loop. A loop is a series of commands that repeats a number of times until a given condition is met. Loops are good for doing repetitive tasks in one script. In our case, we want to move some objects. The loop statement we will use is the "`for`" loop. Imagine that you have a jar with 20 jelly beans. Your boss comes over and says, "I want you to draw a circle on this piece of paper. Every time you do, eat a jelly bean. When you have eaten the last jelly bean, stop drawing circles." So, in a few minutes you have a piece of paper with 20 circles on it and a tremendous sugar rush. You also have grasped the fundamental concept of a `for` loop.

```
// move the objects a random amount
for ( $n = 0 ; $n < `size $selectedObjects` ; $n = ( $n + 1 ) )
   {
    $randomX = `rand -100 100`;
    $randomY = `rand -100 100`;
    $randomZ = `rand -100 100`;
    move -relative $randomX $randomY $randomZ $selectedObjects[$n];
   }
}
```

We initialized the loop at 0, implicitly defining a variable named `$n`, which merely increments by one each time the loop executes. The loop runs until `$n` is no longer smaller than the number of objects in our array `$selectedObjects`. We do this because in MEL, the first object in an array is indexed with the number 0, the second with 1, and so on. This is a little confusing, but as with all things in programming, you quickly become accustomed to it.

Each time the loop runs, we calculate a random number between –100 and 100 for each of our three axes, and we then move the object by that distance, relative to its current position.

That's the end of our script, and we end the procedure with the closing bracket. Save out your script, select a bunch of objects and then type "`randomMove`" at the Maya command line. Because Maya has no clue what `randomMove` is, it will look for a file called randomMove.mel. Once Maya finds it, it executes the procedure `randomMove`, where your objects are randomly spread around. Congratulations, you are now a scripter.

Project 3: Custom Tools

Now that you see that scripting isn't such a big deal, you want to take on something bigger, I'm sure. Why not create a custom tool? Tools are useful things to know how to write, because they allow you to extend the functionality of your art package, and

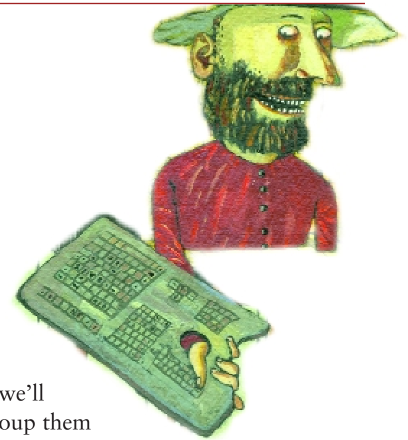
Leveraging Lightwave 1.5

Newtek's Lightwave 1.5 supports a C-like scripting language known as LScript. Similar to the other packages, Lightwave has a log window (the LSCommander) which can be revealed to show what commands are being executed as you work. You can take commands from this window and quickly and easily tie them to buttons in the interface. LSCommander is part of the Layout portion of Lightwave.

Lightwave ships with a suite of tools that simplify the task of designing and debugging scripts. A fully context-sensitive text editor and interface designer are part of this environment (the LSIDE). Scripts can be written in LSIDE, or exported from the LSCommander window, which automatically converts the commands to LScript form. After creating a script, you can compile it down to a platform-independent format which enables you to distribute your script without worrying about someone tweaking the source.

The LScript core engine can link directly to Windows DLLs or Unix shared libraries, so you can use external routines which have been written in C or C++. This allows you to arbitrarily expand the functionality of your interface through a combination of LScript and traditional compiled code. Newtek is in the process of creating an LScript guide to ease script construction for artists.

— Mark DeLoura



do so in a way that works exactly how you need. We will create an array tool, something that will automatically create a radial array of objects from a selected object, and give it some dynamic properties.

First, our header:

```
// radialArray.mel
// Creates a dynamic radial array node.
// Author: David Stripinis E-mail: david@factor5.com
// For Maya 3.0
```

Next, we need to declare the procedure, but this time we want the user to pass some options to the script. In Maya, we do this by including variable declarations in the parentheses after the procedure name. Here, we declare an integer that will define the number of duplicates to create in the radial array.

```
global proc radialArray ( int $numberOfDuplicates )
{
```

Next we declare a variable and define it, all in one statement. This is a good practice that usually makes the script easier to read. Notice that although I only want one object name, I still have to declare a string array as the variable type. It doesn't matter if your list has thousands of items or just one, it is still a list. Let's also perform an error check to make sure the user has selected only one object. Rather than conducting the test "If the size is one, perform some actions," we will use "If the size is not only one object, then stop!"

```
string $selectedObject[] = `ls -sl`;
if ( `size $selectedObject` != 1 )
    error "Please select just one object.";
```

Now that we have the selected object, we need to be able to control its transforms, so we should group it. While this may seem excessive, it guarantees we will have complete control over our object's relative position, without messing up its current transformations. We also retrieve the new selected group and store it in the `$arrayGroups` variable.

```
group;
xform -os -piv 0 0 0;
string $arrayGroups[] = `ls -sl`;
string $temp[];
```

Now that we have a clean transform node and know what it's called, we can begin to work. We also declared a temporary variable that we'll use to hold data in our upcoming loop. This loop will repeatedly duplicate the object until we have the number entered by the user with the `radialArray` command. For instance, if we entered `radialArray 5`, we need to create four duplicates. In this loop we'll use a shorthand notation for incrementing a variable. Rather than typing `$n = ( $n + 1 )`, you can just type `$n++`.

```
for ( $n = 1 ; $n < $numberOfDuplicates ; $n++ )
    {
     duplicate -rr;
     $temp = `ls -sl`;
     $arrayGroups[$n] = $temp[0];
    }
```

Now we have a string array, `$arrayGroups`, which contains the names of all of our objects, including the original selected object. Next we'll select all of the objects and group them once again, so that we have one node that controls the entire array. We'll name that node "`array_group_master`." As you can see, scripts frequently perform the same actions over and over:

```
select -r $arrayGroups;
group;
rename `ls -sl` "array_group_master";
string $masterControl[] = `ls -sl`;
```

We will now use a feature unique to Maya, dynamic attributes, to create some controls. In other packages, you may have to create auxiliary objects to set up similar controls. We add the attribute `Sweep`, which allows us to control how many degrees of a circle the array should cover.

```
addAttr -ln Sweep -at double -min 0 -max 360 $masterControl[0];
setAttr -e -keyable true ( $masterControl[0] + ".Sweep" );
setAttr ( $masterControl[0] + ".Sweep" ) 360;
```

We now simply set the Y-axis rotation for each duplicated object to an expression that represents its angle. What we will do is build a string that contains our expression and then apply that expression to each object. We do it this way simply for clarity.

```
string $raExpression;
for ( $n = 0 ; $n < `size $arrayGroups` ; $n++ )
    {
     $raExpression = ( $arrayGroups[$n] + ".ry = (( " +
         $masterControl[0] + ".Sweep/" +
         $numberOfDuplicates + ") * " + $n + ")" );
     expression -s $raExpression -o $arrayGroups[$n] -ae 1 -uc all ;
    }
}
```

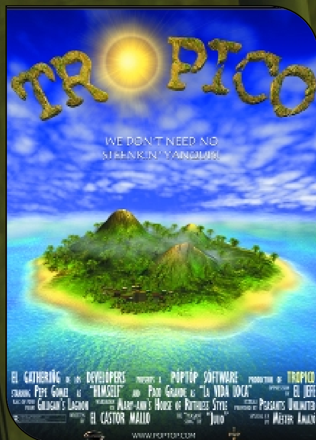
Now we're done. Admittedly, this tool could be expanded quite a bit, but at the beginning, it is better simply to get the tool to a working stage and add new features from there.

Get to It

Well there you have it, the basics of scripting. Hopefully, this article has taken some of the mystery and confusion out of the process. Scripting really isn't a black art or any kind of trade secret. Like everything else, it is simply one more tool in your arsenal as an artist. Happy scripting!

NOTE: As we went to press, Maya 4 was shipping (see Product Reviews, page 8). While the great majority of the time the new version will make no difference for scripting, some commands may have been slightly altered. If problems arise, check your documentation.

Poptop Software's TROPICO



GAME DATA

PUBLISHER: Gathering of Developers

NUMBER OF FULL-TIME DEVELOPERS: 10 (7 artists, 3 programmers)

NUMBER OF CONTRACTORS: 1 musician

ESTIMATED BUDGET: $1.5 million

LENGTH OF DEVELOPMENT: 2 years

RELEASE DATE: April 2001

PLATFORMS: Windows 95/98/ME/2000/NT 4, Macintosh

DEVELOPMENT HARDWARE (AVERAGE): 550MHz Pentium IIIs with 512MB RAM, 40GB hard drives, and a variety of 3D cards running Windows ME or 2000 (programmers) or Windows NT (artists)

DEVELOPMENT SOFTWARE: Visual C++ 6.0, Visual SourceSafe 6.0, 3DS Max 3.1, Character Studio 2.2, Photoshop 5.5, Tree Factory plug-in for 3DS Max

NOTABLE TECHNOLOGIES: Bink Video, Miles Sound System

PROJECT SIZE: Approx. 150,000 lines of code (plus 20,000 for tools)

In the spring of 1999, Poptop had just wrapped up development on the successful RAILROAD TYCOON II (RT2) and its expansion, THE SECOND CENTURY. At the time, Poptop was staffed by the overwhelming count of four artists and two programmers. Being that small, we had had no time to think about anything other than the current project, and suddenly we found ourselves sitting around a table, eyes still slightly glazed from the inevitable project-end rush we had just gone through, looking for new ideas.

These were uncharted waters for Poptop. RT2 had been based very closely on Sid Meier's classic RAILROAD TYCOON. The original RAILROAD TYCOON had been an inspiration, a design manual, a blueprint for making a good game, and a launching point for new ideas. The upside was that a good part of the design work had been done for us. The downside, as we were to find out on our next project, was that it left us a bit naïve about the effort it would take to create a new game from an original idea.

As we sat around our company card table, brainstorming ideas, one idea quickly jumped to the forefront. The idea of taking a building game and putting a political game on top of it had captured everyone's imagination. With our creative energies renewed by a fresh idea and the thrill of starting a new project in our hearts, we rushed off to create TROPICO — each of us in our own way.

Actually, it wasn't that bad. We did discuss major elements of the game. We knew that it would have buildings and people that the player would not control directly. We knew it would use the RT2 engine but would be more ambitious than RT2 had been. As we rushed off to begin development, that was about all we knew — and, as we were to discover later, each person on the team didn't even share the same vision about the things we thought we did know.

During the project, Poptop grew to the bloated size of 10 employees — seven artists and three programmers. It is a testament to the talent and hard work of this team that we ended up with a strong, fun product in spite of the pitfalls that we encountered along the way.

BRENT SMITH | *Brent started working on games in 1992 at Capstone (THE DARK HALF, WAYNE'S WORLD). After a couple of years in the "real" world, he went to Interactive Magic, where he worked on BRUCE JENNER'S WORLD CHAMPIONSHIP DECATHLON, HARPOON CLASSIC 97, and MALKARI. He joined Poptop in early 1999 and has spent the last two years working on TROPICO.*

ISLA PARAISO
BANCO
VIVA
EL
PRESIDENTE
UNION
DE
TRABAJADORES
HECHO
EN
TROPICO
¡VIVA!
TROPICO
TAXI

What Went Right

1. Created "deep" characters. It was obvious from the beginning that the most important aspect of TROPICO would be the people. If we intended to have a game in which the player didn't have direct control of the units on the map, then we had better make sure that the people acted in a reasonable and somewhat predictable fashion.

This was no trivial task. Each unit on the map (in the later stages of the game there can easily be more than 500 units) has more than 70 characteristics, which determine its actions and reactions to the player. This includes items as simple as name, age, and what part of the island the unit was "born" on, to things as complex as the unit's satisfaction with various aspects of the environment (religion, national pride, pay compared to others in the same situation, and so on). Additionally, as units live their sim lives — they are born, prance about as children, enter the workforce at a certain age, and eventually retire and die — we keep track of their families. Each unit potentially has a mother, father, spouse, and multiple children. Units also know about their grandparents, cousins, aunts, and uncles. What this means to the player is that the repercussions for treating one unit badly filter through their family tree much like you would expect in real life. Send Juan Pablo Ramirez to jail, and not only are his parents, wife, and children upset, so are his five siblings, their 18 children, and so on down the line. This added a lot to the political atmosphere of TROPICO.

Our second goal with this system was hiding its complexity from the player. Part of the fun of TROPICO is trying to see and understand what response your actions will evoke from the population of your island. Although we kept detailed information about each unit in the game, there was a balancing act in taking advantage of it without flooding the player with information. I think we succeeded here. Our interface provides the player with in-depth information about the people in the game — making them come alive — without overwhelming the player with having to know trivial details about them.

Unit development was not easy, but because we identified this as a critical area up front and spent a lot of time and manpower addressing it, it turned into one of the strengths of TROPICO.

2. Small, streamlined, and talented team. Being as small as we are certainly has a downside, the most serious being that we are limited in the things we can do in a given amount of time by sheer lack of manpower. However, the advantages to a team this small, at least in terms of developing a project such as TROPICO, outweighed the disadvantages.

Everybody at Poptop knows everyone else. Not just knows them, but knows what they are working on, what their strengths and weaknesses are, the name of their significant other, and so on. There's no hiding here. If you screw up, people know it was you. If you do something brilliant, everyone knows that too.

This kind of intimacy makes us very

ABOVE. An overview of the town. LEFT. Wireframe and model of the hotel. RIGHT. A tourist.

streamlined. Everybody knows where he stands and what his job is. There is no middleman; if you need to talk to someone, you talk directly to him. There is no distraction of having team members pulled off to work on a different, behind-schedule project or promotional material. We did one thing, TROPICO, and that's all we did.

Of course, this kind of team only works if every member is talented, and that is, without a doubt, the case at Poptop. I see it every day, and I think the players of RT2 and now TROPICO have seen the result as well.

3. Homegrown tools. Upon the completion of RT2, we had accumulated a nice suite of tools for preprocessing and manipulating art assets and interface elements. With TROPICO, we continued this work, both improving the existing tools and developing new ones.

Our most-used tool was a program written to preprocess art assets into our custom format. It also had the ability to clip and scale the art, and also reduce animations to a keyframe and delta information for efficient storage. When TROPICO began, we further modified this tool to allow it do work with 24- and 32-bit TIFF files. A palette reducer/optimizer was added to create 8-bit palettes from one or more higher color-depth images. We also included the ability to add parameters to the instruction set that the program used to process the files, allowing such things as tinting (which allowed us to construct placeholder art rapidly by simply taking an existing image and tinting it to another color), and lightening or darkening of the image. Finally, we added support that allowed us to read image-depth information stored in RLA files and store it with the image. This information could then be used to tell us the Z-order information of the various parts of the image, which allowed us, for example, to handle units walking behind parts of a building while walking in front of others.

Another tool that we inherited from RT2 and improved upon during the development of TROPICO allowed us to create, size, and position interface elements outside of the code. Using a simple scripting language, interfaces could be built and then compiled by this tool into a format that could be used by the TROPICO code.

A new type of tool that we developed and began using with TROPICO, and which turned out to be a real time-saver, was used for game data manipulation. Using MFC (which I'd never recommend for any software intended for release, but which is tremendous for quick development of tools such as these), we quickly built a very robust unit editor and building editor. These editors allowed us to manage all the data associated with a particular type of building or unit outside of the program. This capability was invaluable for balancing and tweaking the data, as it allowed us to change information in a relatively safe way while the game was running and see its effects immediately upon the game world.

4. Fun topic. Without a doubt, one of the key factors in the success of TROPICO was the topic. During our brainstorming sessions, a number of ideas were thrown on the table, but the idea that became TROPICO was the one that had everybody excited. While a lot of the elements of TROPICO can be found in other games, the mixture of those elements and the setting itself had everyone eager to see what we could make. This enthusiasm translated outside of the company, too. Nearly everyone to whom we showed the game voiced their enthusiasm about the

TOP LEFT. The dictator's office. BOTTOM LEFT. View from the dictator's patio at sunset. RIGHT. The dictator's mansion.

freshness of the idea. Something about the idea of ruling an island full of sun-drenched beaches and tropical beauties strikes a chord in most people's hearts.

One of the most promising indicators late in the project that we had something special on our hands was that we were still eager to play the game, and were still throwing out new ideas, even after spending two long years developing it.

5. Localization. Having been involved in the translation of RT2 into a variety of different languages, including double-byte handling for Asian languages, we knew up front that this was something that we needed to be concerned with in TROPICO.

Fortunately, a lot of the groundwork had been established during RT2's development. The code contained home-grown string manipulation functions, which allowed us to maintain tight control of many of the issues associated with localization. We also had a tool that allowed us to pull strings out of the code base for insertion into a string table near the end of the project. The tool was very useful in that it not only recognized strings within the code, but it was smart enough to disregard strings within comments and strings on lines which we tagged with a special comment telling the tool that it was O.K. for this string to remain in the code (format strings, for example).

This meant that for much of the project we didn't have to concern ourselves with trying to keep a string table up to date, but when the time came, we were able to do it quickly and efficiently. Overall, localization was a breeze. Having seen what a nightmare this step can be on other projects, we were dreading the work we thought we would have to do in this area, but the final tally was less than a man-week of work spent getting the game ready for foreign language translation.

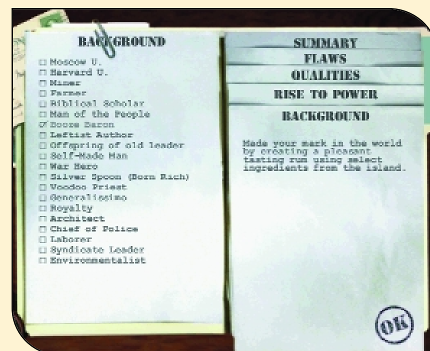
What Went Wrong

1. Lack of up-front design work. Coming off the success of RAILROAD TYCOON II, we were excited to get the chance to work on something new and unique. A few company brainstorming sessions and a game or two of Junta, and we knew that we wanted to do a tongue-in-cheek political game. We couldn't jump into the project fast enough. Ideas were flying hot and heavy. Everyone was excited.

Dictator.

Unfortunately, we failed to realize at the time that everybody was carrying a slightly different picture in his head of what the final game would be. Some were envisioning the stab-your-neighbor, laugh-a-second antics of the Junta board game. Others were seeing the close-up, detailed view of people in action that ROLLERCOASTER TYCOON had done so successfully. Most of us were somewhere in between.

Having come from RT2, we really found ourselves unprepared for this problem. With RT2, Poptop had the original as a blueprint and design document. Design was only an issue so far as how the original game could be improved upon and how current technology could be utilized to improve the game. The game could practically write itself, with



LEFT. An in-game overview of the island. TOP RIGHT. Players create their in-game identity. BOTTOM RIGHT. The almanac.

Poptop's main concern being to re-create the magic of the original game while adding a few minor twists of our own.

Now we found ourselves with a blank slate, an original idea where every gameplay detail had to be created from scratch. Unfortunately, we approached this in much the same way as we had approached RT2. Rather than settling on a unified design, or even trying to create one, each of us ran back to his workstation and began to create what we thought the game would be.

It quickly became apparent that we were not all moving in the same direction. People had very different views of where the game should go. Decisions had to be made on the fly. Some people's visions were cut out entirely, while others had theirs altered to the point that it became something entirely different.

This process was a very difficult growing pain for Poptop. People's feelings were hurt when they realized the idea that they were so excited about was not the idea that we were creating. The game lost "buy-in" from people in the company as it became something that they were less interested in, or someone else's idea that they didn't really understand.

During this time we struggled onward, trying to create this game that wasn't really what anyone had originally intended. Amazingly, we stubbornly refused to stop and consolidate our ideas in meetings or on paper so that the team could be unified in the idea and to rekindle the original excitement for the game. Eventually, working on TROPICO stopped being a passion and became just a job for many on the team, leading to low morale and loss of productivity.

Hopefully, we've learned from this mistake, and on our next idea (original or not) we will figure out what it is we are creating before we start to create it and try to keep everyone excited about the direction in which we are moving.

2. Modifying the existing code base. One of the givens, decided before any work was even begun on TROPICO, was that we would use the 3D engine from RT2. This would allow us to have working maps with many of the features we would need in TROPICO almost immediately, thereby giving us a huge jump start on development.

Pit boss.

The idea was a great one and paid huge dividends in getting us working almost immediately on the game itself rather than the engine. Unfortunately, in conjunction with this decision, we started from the existing RT2 code base — not only the engine, but all the game code from RT2 as well. We were trying to "morph" RT2 into TROPICO, which led to all kinds of problems and slow-downs.

First of all, a single programmer had developed almost the entire RT2 code base. With TROPICO, the staff of Poptop had nearly doubled in size. Each new programmer immediately faced the daunting task of getting a grasp of the RT2 code before he could even begin to make the modifications necessary to create TROPICO.

Second, there were huge chunks of the code base that became dead once TROPICO was started, such as the multiplayer code. (We had decided pretty early in TROPICO's development to scrap multiplayer and concentrate on the single-player experience.) Of course, multiplayer code was integrated very tightly into many areas of the RT2 code, and at first we tried to

work around it. Eventually we tore it out, once we became frustrated trying to determine which areas of the code were dead and which were important.

Finally, the process of working off the original RT2 code base was inherently dangerous at best. We automatically inherited any bug that had managed to survive in RT2 and created quite a few more just going through the process of weeding out what code was unnecessary for TROPICO.

We would have been much better off starting with a clean slate and then pulling over those sections of the RT2 code base that we could use. The RT2 code was not cleanly delineated between engine code and game code. However, the process of untangling the usable engine code and importing it into a clean code base would have been inherently more bug-free and more comprehensible to those unfamiliar with the RT2 code, and would have saved us time in the long run.

3. Fun factor versus gee-whiz factor. Because we started with an existing engine, one of the errors that we made during development was to see how far we could push the envelope with the engine, working on "gee whiz" enhancements that would improve the look and the technology of the game instead of features that would enhance gameplay.

The biggest example of this was what we dubbed "Zoom 0." As the graphics in TROPICO were much more detailed than RT2's, we looked for ways to show off these gorgeous images in the game. Allowing the engine to zoom in one level closer than it had previously been able to (Zoom 1) was one of the ways that we did this. In TROPICO, players can zoom in very close and get very detailed views of the people and the buildings.

Unfortunately, Zoom 0 is not very useful for gameplay, as it is almost impossible to see enough of the map at that zoom level to get a feeling for how you should play. The majority of players tend to stay zoomed out about two levels, occasionally zooming in or out one level as the situation warrants.

O.K., so we added a feature that allowed us to show off the graphics even if it didn't help gameplay. What's the big deal? The deal is that we pre-scaled all of the images for the various zooms beforehand and stored them in the data file, so these high-resolution close-up graphics ate up as much space as all the other zoom levels' graphics combined. We spent a full 50 percent of our graphics budget on this one feature. As we got deep into the project, it became apparent that memory and CD file space budgets were going to be tight, but we had invested too much into this feature to be comfortable with cutting it. Ultimately, we had to cut other features to create space, features which would have improved the game. Rotatable buildings, more unit animations, and repeating animations on the buildings (such as blinking lights and moving machinery) all had to be cut to make room.

Tourists relax and sunbathe on the beach.

Looking back, it is apparent that tossing out Zoom 0 and putting in more gameplay-friendly features would have been a big net improvement to the overall game.

4. Waited too long for scenarios. After the scenario-intense RT2 and its add-ons, we were more than happy to try creating a game in which the strengths lay in randomly generated maps and sandbox-style open-ended play. From day one we worked on TROPICO with this goal in mind. A lot of effort went into creating a map generator that would create pleasing, logical, and most importantly playable maps. We modeled rainfall on what we knew and could find out about meteorology. We researched vegetation, not only to find flora indigenous to a Caribbean setting but also to find out under what conditions a particular plant would thrive and how to represent that in the game. A significant amount of time went into creating realistic mountains and series, even ranges, of mountains. We even went so far as to model the terrain under the water, so that shallow beach areas and deeper waters would be accurately created.

As we neared the end of the project, we had no doubt that our map generator could create some fabulous-looking maps, and, given the number of factors we allowed the player to tweak during map generation, an endless supply of playable maps. However, the game was missing clearly defined goals. Using the map generator, there was no way to create a map that had a specific situation to solve, and only a very limited way to create maps with unique obstacles to overcome. In other words, the game and the maps were great for an open-ended, sandbox style of game, but were lacking in goal-oriented, problem-solving gameplay. While the sandbox mode allows players to create a wide range of different maps, much more depth and many hours of play could have been added to the game if we had included a rich set of scenarios and the tools for the players to create even more.

As we realized this late in the project, we scrambled to create scenarios to include. Unfortunately, our tools in this area were underdeveloped, and time had to be squeezed out of people's schedules even to produce what we did. The result was that TROPICO included a very limited number of scenarios (eight with the game and two others included in some promotional CDs) that weren't nearly as involved as they could have been, and certainly weren't up to the standard that we had created in RT2.

Another by-product of this oversight was that we never spent time polishing the map editor tools that we used in development. The original game design was oriented toward random-map play, so we never saw the need for a more sophisticated editor until late in the project. These tools ended up being disabled in the release version, disappointing many fans who were hoping to

create their own maps. Fortunately for our fans, a map editor should be available in an upcoming patch.

5. Lack of unified artistic vision. As I mentioned, one of the effects of essentially bypassing the design phase of this project was that there was a lack of consistent vision among team members. One of the places that this became most apparent was in the game art.

Almost all of the artists at one time or another during the project worked on creating buildings for the game. They were given only a vague notion of what type of building they were to create — a paper sketch or a very crude 3D mockup — and left on their own to move forward from there. Halfway through the project, the problems with this became apparent. Scale varied wildly from artist to artist, as did level of detail.

Banker.

The same problems were occurring with the character animations, as the two artists working on those had taken different approaches. One artist was striving toward very lifelike figures with complex animations, while the other created more cartoonish parodies of TROPICO's inhabitants with more outlandish but less complex animations. Both artists had a clear idea of what they were trying to do, and both accomplished their respective goals brilliantly, but the difference in their approaches can still be seen in the release version of the game if you compare, for example, the banker to the female luxury tourist.

Luxury tourist.

At this point we appointed a person to take on the role of art producer. His job was to try to coordinate the artists' efforts and make sure they shared more or less the same vision. But the damage had been done. Team members had to spend valuable time sifting through and reworking art.

Frustrations mounted as artists who previously had been able to work toward their personal vision now found themselves having to compromise to a shared vision of the whole team.

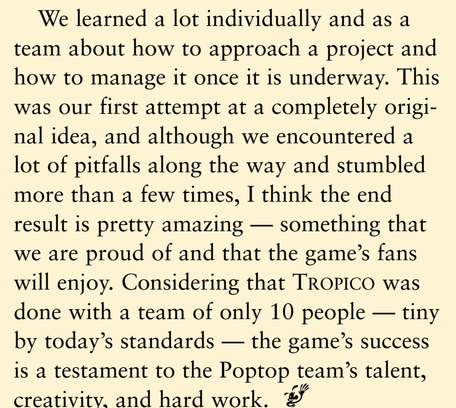
This problem could have been significantly reduced with more up-front planning and more ongoing feedback to the artists as they completed each task. We definitely learned from this process and will improve upon it in our next game.

In Hindsight

It's pretty much the experience with any game project, whether the developers will admit to it or not, that you look back and see mistakes that you made and bemoan the ways that the game could have been better if only those mistakes had been avoided. TROPICO is certainly no different in that regard.

Every member of the TROPICO team felt the sting of loss at some point or another when a feature that they were particularly fond of was cut. Each of us can look back and think of a hundred ways that we could improve TROPICO. That, in itself, is a good sign. At the end of two years of development, we still cared about the game and wanted to make it better. Everyone was happy with what we had created, but no one was satisfied with the details. Art is never done.

We learned a lot individually and as a team about how to approach a project and how to manage it once it is underway. This was our first attempt at a completely original idea, and although we encountered a lot of pitfalls along the way and stumbled more than a few times, I think the end result is pretty amazing — something that we are proud of and that the game's fans will enjoy. Considering that TROPICO was done with a team of only 10 people — tiny by today's standards — the game's success is a testament to the Poptop team's talent, creativity, and hard work.



Beyond Bug Hunting

In game development, no job title is limited to a single role. Designers don't simply design games and a producer's influence isn't limited to management of the game's budget. No one expects those who fill these roles to be strictly confined to them, nor should they be. The game creation process takes a lot more work than the sum of every individual's defined parts.

Among game developers, and even among testers, it's a common misconception that QA's main purpose is simply to test the game that the development team has created. In a perfect world, breaking the game would be the beginning and end of QA's job. In the real world, however, QA's ultimate function can't and shouldn't stop there.

The tester's primary role is to continually assess the game's state so that the development team's priorities can be focused appropriately. Of course, as a result of this assessment, the tester will uncover and report all sorts of bugs, but that's only a means to an end. If a story element doesn't make sense, if a mission objective is contrived, or when a level or enemy is too hard to beat, it's the tester's responsibility to communicate that issue to the development team.

Game developers have a natural tendency to allow their hopes and dreams to influence their opinion of the game. For example, level designers who really want their levels to be a perfect juxtaposition of art, gameplay, and bug-free geometric design may develop that belief over time with little regard for the feedback they receive. The same thing can be said of most other members of the development team, who build so many figurative trees for a project as to lose sight of the forest. For the developer, there's nothing wrong with having a natural biased opinion of the game. It serves as a motivating force to getting the game finished and for that reason alone should be encouraged. For the testers, though, injecting some reality into the developer's assumptions is their primary job.

Illustration by Peter Ferguson

Communicating that perspective is difficult for testers. When the developer is immersed in a dream world where the game is actually coming along well, it's hard to tell him or her otherwise. Sometimes, all developers want to hear from testers is how many bugs are in the game, not necessarily where the overall look, feel, and fun factor of the game might be lacking. Of course, developers can never assume a game is bug-free, but they often assume that the fundamental aspects of the game are solid and therefore immune to criticism.

Testers also run the risk of losing an unbiased perspective. Too often, testers begin to think that they should not only point out problem areas, but also have some part in fixing them as well. That trap must be avoided at all costs. It is not QA's job to fix the problems they report, nor should it be. Getting too involved in fixing the game's problems (as opposed to merely pointing them out) leads to an emotional connection with the game, invariably resulting in the same loss of objectivity that the rest of the development team

continued on page 63

continued from page 64

may naturally suffer from and which the tester was meant to counterbalance. This situation ultimately ends in a game that doesn't live up to its potential.

Some of you readers are testers, and many more of you were at one time. So why do you need to hear all this? In my experience, even testers don't understand the true scope of their job, and the importance that naturally goes along with it. In fact, understandably, most people in the game industry wouldn't want to be in this position (or ever want to be in it again if they've since moved on from testing). After all, constantly pointing out someone else's mistakes is, if nothing else, a depressing endeavor.

Game development is from beginning to end an exhaustingly collaborative process. Working with the QA team is no exception, because the QA process really does extend beyond merely "testing" the game. A tester is the person developers must rely on to dive in, dig deep, and come up (hopefully) with the marrow of the game. Testers, in turn, must try to understand the game the developers envision and be able to compare it objectively with the game they're actually creating. For that reason alone, an open-minded design team would do well to listen to the testing group's opinions and, if appropriate, refocus the team's priorities. The next time you submit a game to QA, consider sitting down next to the tester as he or she plays part of the game. By seeing the game through the tester's eyes, you'll learn more about your game than by just reading through the bug database. In the end, I promise your game will be better for it.

CHUCK MCFADDEN | *Chuck has worked in the LucasArts QA department for three years. In that time, he has worked on games such as RACER, JEDI POWER BATTLES, and BATTLE FOR NABOO. He is currently lead-testing STAR WARS ROGUE LEADER for the Nintendo Gamecube.*